1985

The design and analysis of a complete entity-relationship interface for the Multi-Backend Database System.

Goisman, Philip L.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DESIGN AND ANALYSIS
OF A COMPLETE ENTITY-RELATIONSHIP
INTERFACE FOR THE
MULTI-BACKEND DATABASE SYSTEM

By
Philip L. Goisman

December 1985

Thesis Advisor: David K. Hsiao

URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION: val Postgraduate School 6b. OFFICE SYMBOL (If applicable): 52	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
ADDRESS (City, State, and ZIP Code): nterey, CA 93943-5100	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5100
NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

TITLE (Include Security Classification)
The Design and Analysis of a Complete Entity-Relationship Interface for the Multi-Backend Database System

PERSONAL AUTHOR(S)
Philip L. Goisman

TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
aster's Thesis	FROM _____ TO _____	December 19, 1985	93

SUPPLEMENTARY NOTATION

COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Multi-Lingual Database System, Multi-Backend Database System, Entity-Relationship Model, Daplex

ABSTRACT (Continue on reverse if necessary and identify by block number)

Interest in increasing programmer productivity has spawned new software tools. Some of these tools are statistical packages, program generators, and database management systems (DBMS). In the area of DBMS, research is ongoing to improve the efficiency of DBMS tools. One research effort to improve the efficiency of DBMS is the multi-lingual database system (MLDS). MLDS combines softare and hardware technology to gain efficiency and versatility in DBMS. The MLDS design goals overcome the conventional limitation to develo a database system that supports a single data model and a correspond-ing model-based data language. Examples of data models are relational, hierarchical, network, and entity-relationship. Examples of correspond-ing model-based data laguages are SQL, DL/I, CODASYL, and Daplex. These models and their data languages are supported conventionally (Cont)

DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
a NAME OF RESPONSIBLE INDIVIDUAL Prof. David K. Hsiao	22b TELEPHONE (Include Area Code) 408-646-2253 / 22c OFFICE SYMBOL 52Hq

D FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

ABSTRACT (Continued)

by separate DBMS. Instead, MLDS as a single DBMS is capable of supporting multiple models and their respective database languages.

In this thesis we present a methodology for supporting entity-relationship database management on an attribute-based database system, since the heart of MLDS is the attribute-based system. Specifically, we provide the design specifications for transforming Daplex requests into equivalent attribute-based data language requests. During this design process, we describe the data structures, control structures, and the functions required to implement this transformation.

Approved for Public Release, Distribution Unlimited.

The Design and Analysis of a Complete Entity-Relationship Interface for the Multi-Backend Database System

by

Philip L. Goisman
Major, United States Army
B.A., Temple University, 1967

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1985

ABSTRACT

Interest in increasing programmer productivity has spawned new software tools. Some of these tools are statistical packages, program generators, and database management systems (DBMS). In the area of DBMS, research is ongoing to improve the efficiency of DBMS tools. One research effort to improve the efficiency of DBMS is the multi-lingual database system (MLDS). MLDS combines software and hardware technology to gain efficiency and versatility in DBMS. The MLDS design goals overcome the conventional limitation to develop a database system that supports a single data model and a corresponding model-based data language. Examples of data models are relational, hierarchical, network, and entity-relationship. Examples of corresponding model-based data languages are SQL, DL/I, CODASYL, and Daplex. These models and their data languages are supported conventionally by separate DBMS. Instead, MLDS as a single DBMS is capable of supporting multiple models and their respective database languages.

In this theses we present a methodology for supporting entity-relationship database management on an attribute-based database system, since the heart of MLDS is the attribute-based system. Specifically, we provide the design specifications for transforming Daplex requests into equivalent attribute-based data language requests. During this design process, we describe the data structures, control structures, and the functions required to implement this transformation.

TABLE OF CONTENTS

I. INTRODUCTION 9

A. BACKGROUND 10

B. ORGANIZATION OF THE THESIS 10

II. THE MULTI-LINGUAL DATABASE SYSTEM (MLDS) 12

A. THE ATTRIBUTE-BASED DATA MODEL 13

B. THE ATTRIBUTE-BASED DATA LANGUAGE 14

C. THE MULTI-BACKEND DATABASE SYSTEM (MDBS) 16

III. DAPLEX 20

A. THE ENTITY-RELATIONSHIP (E-R) DATA MODEL 21

B. AN OVERVIEW OF DAPLEX 24

1. The DDL Portion Of Daplex 25

2. The DML Portion Of Daplex 29

IV. DATA STRUCTURES NECESSARY TO EXECUTE DAPLEX 31

A. A METHODOLOGY FOR THE DESIGN OF DATA STRUCTURES 31

B. DESIGN OF DATA STRUCTURES FOR DAPLEX 33

V. THE TRANSFORMATION OF DAPLEX SCHEMA INTO ABDL SCHEMA 35

A. THE BACKGROUND AND METHODOLOGY 35

B. THE SPECIFICATION 36

VI. TRANSLATING DAPLEX TRANSACTIONS TO ABDL TRANSACTIONS 40
A. THE CREATE STATEMENT 41
B. THE DESTROY STATEMENT 48
C. THE FOR EACH STATEMENT 49
D. THE ASSIGNMENT STATEMENT 55
E. THE INCLUDE STATEMENT 57
F. THE EXCLUDE STATEMENT 60
G. THE MOVE STATEMENT 63
VII. IMPLEMENTATION SPECIFICATIONS 69
A. PARSING AND TRANSLATING BY KMS 69
B. CONTROLLING BY KC 70
VIII. RESULTS AND CONCLUSIONS 72
APPENDIX A: THE UNIVERSITY DATABASE SCHEMA 73
APPENDIX B: DAPLEX DATA STRUCTURES 78
APPENDIX C: DAPLEX TRANSLATION ALGORITHMS 81
LIST OF REFERENCES 90
INITIAL DISTRIBUTION LIST 92

LIST OF FIGURES

Figure 2.1 The Multi-lingual Database System (MLDS) 12

Figure 2.2 The Multi-Backend Database System (MDBS) 17

Figure 4.1 Set-Theoretic Depiction of Sample Daplex Database 32

Figure 5.1 Template Of ABDL University Database Schema 39

Figure 6.1 The CREATE in Daplex 41

Figure 6.2 A Daplex Sample for the CREATE 41

Figure 6.3a An Equivalent Example of CREATE in ABDL Templates 43

Figure 6.3b An Equivalent Example of the CREATE in ABDL 43

Figure 6.4 Example of DESTROY Statement 49

Figure 6.5 The FOR EACH Loop 49

Figure 6.6 Examples of FOR EACH Loop and Their ABDL Translations 54

Figure 6.7 Examples of The ASSIGNMENT Statement 57

Figure 6.8 Examples of INCLUDE Statement 60

Figure 6.9 Examples of EXCLUDE Statement 63

Figure 6.10 Examples of the MOVE Statement 68

Figure A.1 University Database Schema 75

Figure A.2 Logical Graphical Representation of University Database Schema 76

Figure A.3 Generalization Hierarchy of Person for the University Database Schema 77

Figure C.1 Generalized Mapping Algorithm For CREATE 81

Figure C.2 Generalized mapping algorithm For DESTROY 81

Figure C.3 Generalized Mapping Algorithm For FOR EACH Loop 82

Figure C.4 Generalized Mapping Algorithm For ASSIGNMENT 85

Figure C.5 Generalized Mapping Algorithm For INCLUDE 86

Figure C.6 Generalized Mapping Algorithm For EXCLUDE 87

Figure C.7 Generalized Mapping Algorithm for MOVE 89

I. INTRODUCTION

Interest in increasing programmer productivity has spawned new software tools [Ref. 1]. Some of these tools are statistical packages, program generators, and database management systems (DBMS). These tools, although designed to increase programmer productivity, strained hardware capabilities and spawned many new software products all designed to increase programmer productivity. In the area of DBMS, research is ongoing to improve the efficiency of DBMS tools. This research ranges from software solutions specially designed for particular hardware to specially designed hardware systems for data management called database machines.

One research effort to improve the efficiency of DBMS is the multi-lingual database system (MLDS) [Ref. 2]. MLDS combines software and hardware technology to gain efficiency and versatility in DBMS. The MLDS design goals overcome the conventional limitation to develop a database system that supports a single data model and a corresponding model-based data language. Examples of data models are relational, hierarchical, network, and entity-relationship. Examples of corresponding model-based data languages are SQL, DL/I, CODASYL, and Daplex. These models and their data languages are supported conventionally by separate DBMS. MLDS as a single DBMS is capable of supporting multiple models and their respective database languages. Other design theses on the design and analysis of various model interfaces for a single DBMS are on the hierarchical [Ref. 3, 4], network [Ref. 5], and relational [Ref. 6] models. MLDS transforms traditional database models into a single database model called the attribute-based model [Ref. 7]. The attribute-based data model is the heart of MLDS. Implementation theses of various model interfaces for the multi-lingual database system are based on the following corresponding model-based data languages: SQL [Ref. 8], DL/I [Ref. 9], Codasyl-DML [Ref. 10], and Daplex [Ref. 11]). MLDS translates these data languages into a single data language called the attribute-based data language (ABDL).

A. BACKGROUND

Just as an operating system supports a wide variety of data structures and programming languages, MLDS supports a wide variety of database models and database languages. Similarly, as an operating system has many modes of access, such as interactive or batch processing, MLDS provides many modes of access, such as individual query or multiple transaction processing. The final analogy between operating systems and MLDS is that they are both invisible to the user. In other words, MLDS supports the user's chosen data model and language without making the data conversion and language translation known to the user, i.e., a database user inputing SQL transactions will have the answer returned in relational form. This allows users the latitude of continuing to use conventional models and languages which they are familiar with, while developing expertise in MLDS which offers advantages over the systems they previously use.

To date the database models and languages that can be transformed and translated are:

(1) The hierarchical database model and IBM's Data Language I (DL/I) (supported conventionally by IBM's Information Management System, IMS);

(2) The relational model and IBM's Structured English Query Language (SQL) (supported conventionally by IBM's SQL/Data System);

(3) The network model and Univac's CODASYL Data Manipulation Language (CODASYL-DML) (supported conventionally by Univac's CODASYL-DML/Data System); and,

(4) The entity-relationship model and CCA's Daplex Language (supported conventionally by CCA's Daplex/Data System).

This thesis concentrates on the design and analysis of an entity-relationship language interface for the multi-lingual database system, i.e., aforementioned item (4).

B. ORGANIZATION OF THE THESIS

Chapter 2 presents an overview of the MLDS which includes an overview of the attribute-based data model and data language. In Chapter 3 an overview

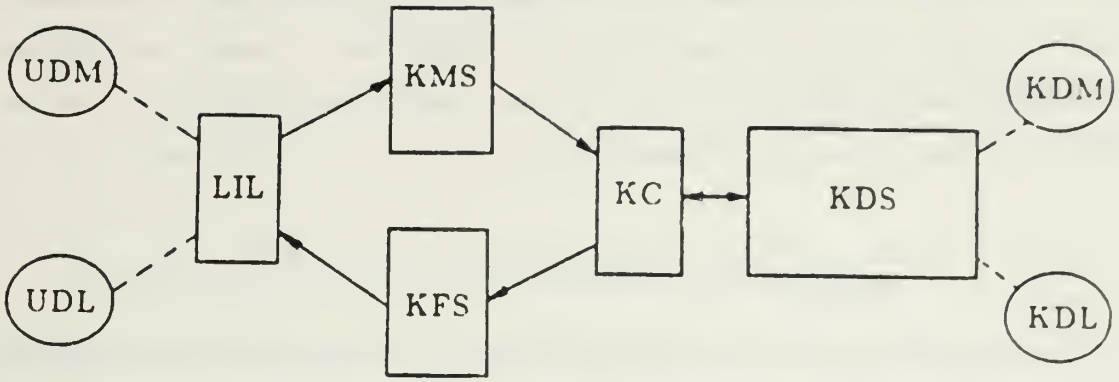
of Daplex is presented which includes an overview on the entity-relationship model, the basis of Daplex. Chapter 4 describes the data structures required by the language interface to translate Daplex transactions to ABDL transactions. Chapter 5 describes the transformation of the Daplex schema into the ABDL schema. Chapter 6 describes the Daplex transaction statements and control commands and the method in which they are translated into ABDL transactions. Chapter 7 describes the proposed specification for the expansion of the design of data structures into implementation of the data definition language (DDL) of the Daplex grammar using pseudo YACC. It also describes the expansion of the design of translating transactions into the data manipulation language (DML) of the Daplex grammar using pseudo YACC. Finally, Chapter 8 summarizes the conclusions from our research experience.

II. THE MULTI-LINGUAL DATABASE SYSTEM (MLDS)

MLDS consists of the modules as shown in Figure 2.1. The user data model (UDM) and the user data language (UDL) refer to the actual database model and language used by a user, e.g., entity-relationship model and Daplex language. The transformation and translation for each database model and its corresponding language is handled by the four modules: the language interface layer (LIL), the kernel mapping system (KMS), the kernel formatting system (KFS), and the kernel controller (KC). These modules together are referred to as the *language interface*. There is a set of these modules for each conventional database system replaced. There are SQL, DL/I, CODASYL-DML, and Daplex interfaces. The kernel database system (KDS) executes transactions from the kernel data language (KDL) for the kernel data model (KDM).

The functions and operations of the language interface are as follows. LIL receives the chosen database definitions and transactions respectively in the UDM and the UDL form and sends them to KMS. KMS transforms the database definition from UDM to KDM or translates the user transactions from UDL to KDL. KMS then forwards the transformed database definition (in the KDM form) or the translated transaction (in the KDL form) to KC. KC sends the transformed database definition or translated transaction to KDS for processing. Upon completion of the specified operation, KC receives the results and sends them to KFS. KFS reformats the results into UDM format and sends the transformed results back to LIL. LIL completes the process by sending the results back to the user.

KDM, KDL, and KDS correspond to the attribute-based model, the attribute-based data language, and the multi-backend database system, respectively. In the following three sections we explore and examine the attribute-based model, the attribute-based data language, and the multi-backend database system.



UDM : User Data Model
UDL : User Data Language
LIL : Language Interface Layer
KMS : Kernel Mapping System
KC : Kernel Controller
KFS : Kernel Formatting System
KDM : Kernel Data Model
KDL : Kernel Data Language
KDS : Kernel Database System

Figure 2.1 The Multi-lingual Database System (MLDS)

A. THE ATTRIBUTE-BASED DATA MODEL

The data structures of the attribute-based data model include: database, file, record, attribute-value pair, keyword, attribute-value range, directory keyword, non-directory keyword, directory, record body, predicates, and query. A *database* consists of a collection of files. Each *file* contains a group of records characterized by a unique set of directory keywords. A *record* has two parts. The first part is a collection of attribute-value pairs or keywords. The *attribute* of an attribute-value pair defines a specific quality or certain characteristics of the value, e.g., in the example of file "Person" below, the "name" is an attribute. Each record can only have one value associated with a corresponding attribute in the *attribute-value pair*. Further, no two attribute-value pairs have the same attribute in a record, i.e., all attributes are distinct in a record. Certain attribute-value pairs of a record are *directory keywords* since their attribute values or attribute-value ranges are kept in a directory, e.g., <FILE, Person>

may be in the directory where "FILE" is the attribute and "Person" is the value of the attribute. The rest of the attribute-value pairs are *non-directory keywords*. The second part of the record is the *record body* which contains only textual information. An example of a record in the "Person" file is as follows:

(<FILE, Person>, <NAME, Philip Goisman>, <SSN, 209327902>,
{ Major })

Note that each attribute-value pair contains only one value; and all the attributes are distinct. The textual information of the record is contained within the curly braces, i.e., the record body.

A *keyword predicate* of 3-tuples consists of an attribute, a relational operator, and an attribute value, e.g., (NAME = Philip Goisman). A query combines keyword predicates in disjunctive normal form. The following is a query.

((FILE = Person) and (NAME = Philip Goisman)) or
((FILE = Person) and (NAME = Mark Gross))

Without any confusion we note that we use parenthesis for both records and predicates.

B. THE ATTRIBUTE-BASED DATA LANGUAGE

The attribute-based data language (ABDL) [Ref. 2, 12] is defined in this section. ABDL supports the five primary database operations, INSERT, DELETE, UPDATE, RETRIEVE, and RETRIEVE-COMMON. A *request* in ABDL is a primary operation with a qualification. A *qualification* specifies the part of the database on which the request operates. Two or more requests grouped together form a *transaction.* In the remainder of the section each request will be discussed and illustrated with an example.

The INSERT request inserts a new record into the database. The qualification of an INSERT request is a list of keywords and a record body. The following example illustrates the INSERT request.

INSERT (<FILE, Person>, <NAME, Philip Goisman>,
<SSN, 209327902>, <GRADE, 3.0>
{student})

This request inserts Philip Goisman, his social security number and a comment that he is a student into the Person file.

A DELETE request removes one or more records from a database. The qualification of a DELETE request is a query. The following example illustrates the DELETE request.

DELETE ((FILE=Person) and (NAME=Philip Goisman))

This request removes all those named Philip Goisman from the Person file.

The Update request modifies records of the database. The qualification of an UPDATE request consists of two parts, the query and the modifier. The *modifier* specifies how the records being modified are to be updated. The following example illustrates the UPDATE request.

UPDATE ((FILE=Person) and (NAME=Philip Goisman))
(GRADE = GRADE + 1.0)

This request updates the grade one point higher for all persons named Philip Goisman. The query is ((FILE=Person and (NAME=Philip Goisman)) and the modifier is (GRADE = GRADE + 1.0)

A RETRIEVE request retrieves records from the database. The qualification of a retrieve request consists of a query, a target-list, and an optional by-clause. The query specifies which records are to be retrieved. The *target-list* consists of a list of output attributes. It may also consist of an *aggregate operation*, i.e., AVG, COUNT, SUM, MIN, MAX, on one or more output attributes. The optional *by-clause* may be used to group records when an aggregate operation is specified. The following example illustrates the RETRIEVE request.

RETRIEVE ((FILE=Person) and (NAME=Philip Goisman))
(GRADE) by SSN

This request lists the grades by social security numbers of all persons named Philip Goisman. The query is (FILE=Person) and (NAME=Philip Goisman), the target-list is GRADE, and the by-clause is by SSN.

The last request, RETRIEVE-COMMON, is used to merge two files by common attribute-values. Logically, the RETRIEVE-COMMON request can be considered as two retrieve requests that are processed serially in the following

form.

```
RETRIEVE (query-1)(target-list-1)
COMMON  (attribute-1,attribute-2)
RETRIEVE (query-2)(target-list-2)
```

The common attributes are attribute-1 (associated with the first retrieve request) and attribute-2 (associated with the second retrieve request). The following example of RETRIEVE-COMMON from a population census example will illustrate this request.

```
RETRIEVE ((FILE=CanadaCensus) and
        (POPULATION >= 100000))(CITY)
COMMON (POPULATION, POPULATION)
RETRIEVE ((FILE=USCensus) and
        (POPULATION >= 100000))(CITY)
```

This example finds all the records in the CanadaCensus file with population greater than 100,000, finds all the records in the USCensus file with population greater than 100,000, identifies records of respective files whose population figures are common, and returns the two city names whose cities have the same population figures. ABDL provides five seemingly simple database operations, which are nevertheless capable of supporting complex and comprehensive transactions.

C. THE MULTI-BACKEND DATABASE SYSTEM (MDBS)

MLDS allows users to work with their favorite data language without having the database system which supports the data language physically present on the same computer. However, a conventional database system which supports that data language requires the system to operate on the mainframe in competition for memory and peripheral resources with all other database systems and their applications. Although a conventional database system does not need to translate transactions or transform databases it nevertheless, competes with all other database systems and applications such as SQL/Data System and SQL transactions. Thus, the degradation of their performance is due to many database systems and applications that operate concurrently on the same computer. An obvious solution to improving performance of all the database

systems and their applications is to reduce the amount of resource sharing cost-effectively. This solution is to offload the database-system software from the mainframe computer to a separate, dedicated computer with its own disk system. This is called the software single-backend approach.

Bell Laboratories has been first to adopt this idea in developing a database system known as XDMS. XDMS has the goal to:

(1) obtain a cost saving and a performance gain through specialization of the database operations on a dedicated backend processor,

(2) allow the use of shared databases [by different mainframe computers, now called hosts],

(3) provided centralized [i.e., physical] protection of the databases, and

(4) reduce the complexity when developing software for a stand-alone and new machine.

It has been determined that single backends are cost-effective. However, they do not improve the performance of database systems entirely. Instead, the single backend can become the bottleneck with increased use of the database system. The problem of performance degradation due to heavy I/O usage still remained.

Dr. Hsiao's group developed an approach to this problem called the software multi-backend approach. The multi-backend database system (MDBS) uses a parallel architecture of backends controlled by a single backend controller. (See Figure 2.2.) Backends are individual database processors, each with its disk system consisting of at least a disk controller and 1 or more disk drives. The backend controller supervises the execution of the database transactions and interfaces with the hosts and users through three levels of menus. Level 1 is the system level which allows various configurations of multiple backend database systems to be generated and initiated. Level 2 permits the main actions of the user and test interfaces. These actions are generating a database, loading a database, and executing the request interface module, which leads to level 3. Menus in level 3 allow the user to choose a new database, create a list of transactions, modify a list of transactions, select a list of previously defined transactions, or display the results from a list of transactions.

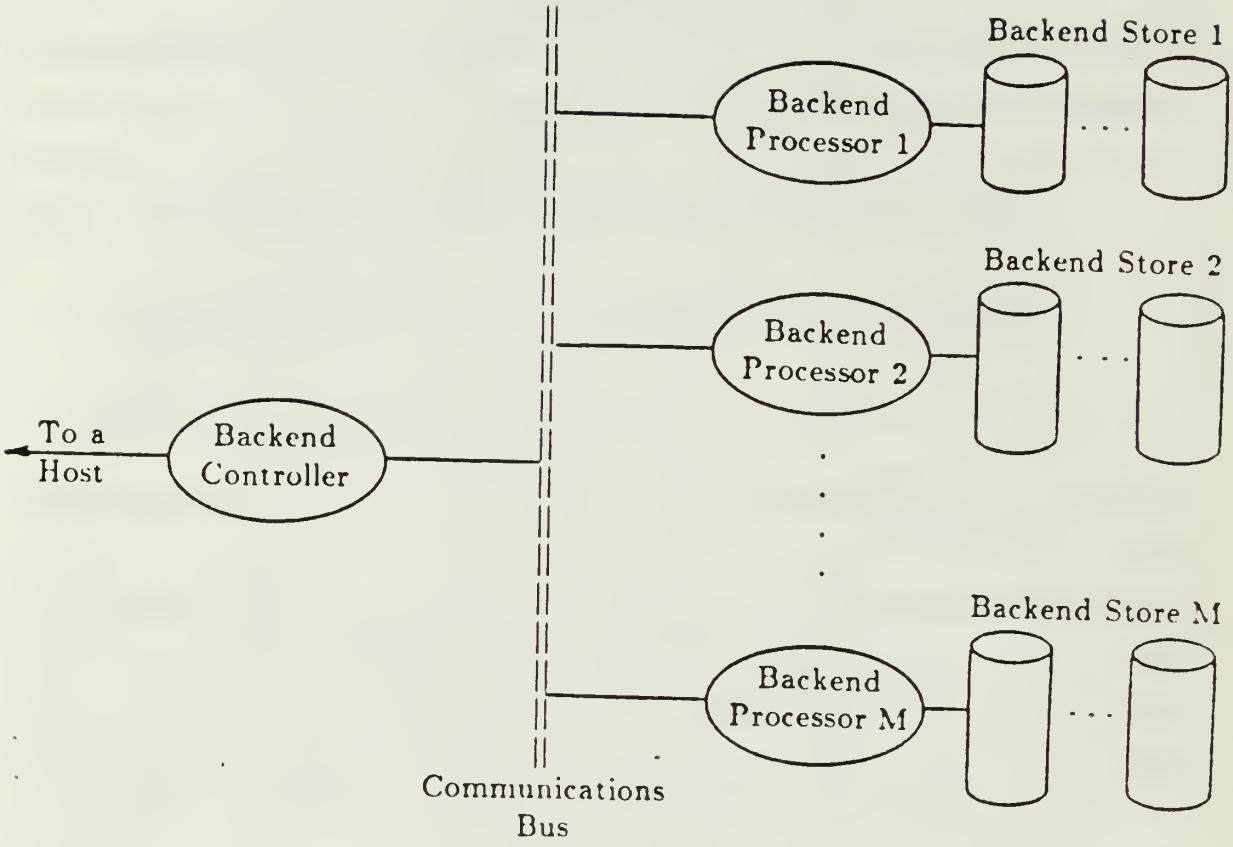


Figure 2.2 The Multi-Backend Database System (MBDS)

When generating or loading a database, the user data is received by the controller and evenly distributed via a communications bus to all of the backends in accordance with a round-robin-like algorithm. During the database loading process, the controller develops a directory in each backend to record the physical location of every record in the database, i.e., on the backend's disk system. The directory of each backend is to be used to locate records for transaction processing.

Each backend develops its own queue of transactions for processing and performs its own database operations. This process is enabled by replicating all the backend software and most of the directory information on each backend. All transactions are sent to each backend. Directories are identical except for the

individual addresses of records (record-ids). Hence, as a backend finishes processing a transaction it sends the transaction responses back to the controller by way of the communications bus.

MBDS has been designed to achieve two performance goals. Goal one states that if we increase the number of backends, while the size of the database and the size of the responses to the transactions remain constant, we can produce a reciprocal decrease in the response times of the user transactions. Goal two states that if we increase the number of backends proportional to the increase in the size of responses to the transactions, we can produce invariant response times for the user transactions.

The first goal demonstrates increased performance in comparison to a single-backend system. The second goal sets out to prove that maximum performance may be maintained even with heavier usage. These design goals also comply with the goal of MLDS, i.e., MLDS must be high-performance with expandable hardware. The interested reader is referred to [Ref. 7, 13, 14] for a thorough and comprehensive discussion of the concepts and capabilities of MBDS.

III. DAPLEX

Daplex is a database language that was created by David W. Shipman [Ref. 15] in 1979 while working at the Computer Corporation of America (CCA) and the Massachusetts Institute of Technology (MIT) [Ref. 16]. Its foundation is in what Shipman and Gray [Ref. 16] call the functional data model. One of Shipman's goals for Daplex is to provide a "conceptually natural" database interface language. That is, the Daplex constructs used to model real-world situations are intended to closely match the conceptual constructs a human being might employ when thinking about those situations. Such conceptual naturalness, to the extent it has been achieved, presumably simplifies the process of writing and understanding Daplex requests, since the translation between the user's mental representation and its formal expression in Daplex is more direct.

Gray notes that Shipman developed his concepts from the semantic net used in artificial intelligence. The *semantic net* is a structure that represents associations between objects. For each object of a given type, there is a corresponding collection of functions which are applicable to it; some of these provide simple values, but the results of others are found by following 'arcs' in the net, which connect the object to other objects of various types. Functions can be applied in turn to these objects, thus exploring a network of associations. Consequently, Shipman's Daplex relies on functions or functional composition to derive actual values.

However, Daplex, as implemented by CCA [Ref. 17, 18], rests more firmly on the entity-relationship model originated by Chen [Ref. 19] and further described by Ullman [Ref. 20] than on Shipman's functional model. We are, therefore, focusing on the entity-relationship model. When application of functional concepts are used, we then note the application.

Throughout this thesis, references to CCA's sample database, sample queries, and graphical database representation will be used in analyzing theoretical concepts or developing design issues. CCA's sample university database and the

graphical representation of the university database are presented in Appendix A. Other examples of the university database will be used appropriately in the chapters as they are needed.

In the remainder of this chapter, the entity-relationship model is first to be presented to provide the basis of understanding Daplex. Then a Daplex overview is to be presented for the purpose of familiarizing the reader with the Daplex schema and transactions which are involved in the transformation and translation of Daplex into ABDL.

A. THE ENTITY-RELATIONSHIP (E-R) DATA MODEL

One of the goals of Chen's entity-relationship (e-r) model is to present a logical view of data. This is an important issue for any DBMS. In conceptualizing any DBMS many questions arise. Two significant questions are [Ref. 20]:

(1) What are appropriate data structures with which to implement a physical database?

(2) What are the properties of typical data and how should it be represented by physical structures?

The e-r model attempts to answer these questions by adopting the more natural view that the real world consists of entities and relationships. The e-r model can achieve a high degree of data independence (i.e., independent from implementation considerations) and is based on the set theory and the relation theory. According to Chen the reader may view the e-r model as a generalization or extension of existing models.

In developing the e-r model Chen identified four levels of logical views of data which apply to the e-r model. They are:

(1) Information concerning entities and relationships which exist in our minds.

(2) Information structure--organization of information in which entities and relationships are represented by data.

(3) Access-path-independent data structure--the data structures which are not involved with search schemes, indexing schemes, etc.

(4) Access-path-dependent data structure.

At the first level we consider entities and relationships. Basically, an *entity* is a thing and is distinguishable. A specific person, company, or event is an example of an entity. Entities are classified into different entity sets such as the sets named Person, Employee, and Student. To belong in an entity set, an entity has the properties common to the other entities in the entity set. Among these properties is a test predicate associated with each entity set to test whether an entity belongs to it. For notational purposes let e represent an entity that exists in our minds and $\mathbf{E}_i$ denote the ith entity set.

A *relationship* is an association among entities. For example, husband-wife is a relationship between two person entities. Relationship is more formally defined in terms of its presence in a relationship set. A *relationship set,* $\mathbf{R}_i$, is a mathematical relation among n entities each taken from an entity set:

$$\{(e_i, e_2, ..., e_n) \mid e_1 \textit{ in } \mathbf{E}_1, e_2 \textit{ in } \mathbf{E}_2, ..., e_n \textit{ in } \mathbf{E}_n\},$$

and each tuple of entities, $(e_1, e_2, ..., e_n)$, is a *relationship.* Intuitively, the *role* of an entity in a relationship is the function that the entity performs in the relationship.

The information about an entity or a relationship is obtained by observation and measurement, and is expressed by a set of attribute-value pairs. *Attributes* are the common properties needed to classify entities and relationships. An *attribute* can be formally defined as a function which maps from an entity set or a relationship set into a value set or a Cartesian product of value sets:

$$f: \mathbf{E}_i \textit{ or } \mathbf{R}_i \rightarrow \mathbf{V}_i \textit{ or } \mathbf{V}_{i1} \times \mathbf{V}_{i2} \times \cdots \times \mathbf{V}_{im}.$$

Values are the qualities of attributes that distinguish entities. Examples of values are Philip, Goisman, 41, 209-32-7902. For the entity set "Person," the above values are values for the Person attributes: first-name, last-name, age, and social security number. Note that an attribute is defined as a function. Also, note that both entities and relationships have attributes.

The entities, relationships, and attributes at level 1 are conceptual objects in our minds. At level 2, we consider representatives of conceptual objects. To organize information associated with entities and relationships, Chen separates the information about entities from the information about relationships. One method of distinguishing amongst a class of entities or relationships is through the use of primary keys or keys. *Primary keys* uniquely identify an entity in an entity set through an attribute or set of attributes of that entity set such that the mapping from the entity set to the corresponding value(s) of the attribute(s) is one-to-one. If we cannot find such one-to-one mapping, we may define an artificial attribute and its value so that such mapping is possible. In Chapter 5 we are going to see that this is precisely the solution we use to transform Daplex data to ABDL data. The artificial attribute we define is a unique key for each entity in its corresponding entity set. For example, in the entity set Person. we define the attribute "Person.key" which is a unique number for each entity.

Since a relationship is identified by the involved entities, the primary key of a relationship can be represented by the primary keys of the involved entities. In certain cases, the entities in an entity set cannot be uniquely identified by values of their own attributes; thus we must use a relationship(s) to identify them. Theoretically, any kind of relationship may be used to identify entities. CCA's graphical representation of their university database in Appendix A uses a one-to-many mapping in which the existence of many entities on one side of the relationship depends on the existence of one entity on the other side of the relationship. As Chen states, "This method of identification of entities by relationships with other entities can be applied recursively until the entities which can be identified by their own attribute values are reached."

In using relationships to identify entities, Chen identifies two forms of entity relations. If relationships are used for identifying the entities, they are called *weak entity relations*. Ullman names weak entity relations *built-in* relationships or *isa* relationships. They are characterized directly by the hierarchical structure amongst entity sets. For example, CCA's graphical representation in Appendix A denotes these as isa relationships. In this example the entity set "student" forms an isa relationship with the entity set "person."

Throughout this thesis we use the term built-in or isa to denote weak entity relations. If relationships are not used to identify entities, they are called *regular entity relations*. Theses relations usually represent themselves, i.e., provide labels or names for themselves. For example, in CCA's graphical representation of their university database in Appendix A, the entity set "course" is a regular entity relation. Similarly, there are two forms of relationship relations. If all entities in the relationship are identified by their own attribute values, they are called *regular relationship relations*. For example, in CCA's graphical representation of their university database in Appendix A, the entity set "person" is a regular relationship relation. If some entities in the relationship are identified by other relationships, they are called *weak relationship relations*. Ullman names weak relationship relations *user-defined* relationships. They are characterized by their roles between entities. For example, in CCA's graphical representation of their university database in Appendix A, in the entities for entity sets "student" and "support_staff," the attributes "enrollments" and "supervisor" specify user-defined relationships. Throughout this thesis we will use the term user-defined to denote weak relationship relations. When distinctions are required between identifications of the relations they will be noted.

Level 3 and level 4 denote the e-r model using a distinct diagrammatic technique. This diagrammatic technique relies on rectangles, diamonds, and arcs developed by Chen. CCA's graphical representation in Appendix A drops this notation. However, it can be seen that if relationships were enclosed in diamonds in the CCA representation, that representation would take on the e-r diagram format.

B. AN OVERVIEW OF DAPLEX

Daplex is a database language used to access and manipulate data modeled in an entity-relationship like model. Database languages usually consist of two different parts, the data definition language (DDL), also, called the schema, and the data manipulation language (DML). This is also true of Daplex. This section introduces the primitive objects, the properties of the objects, and the relations among the objects in the DDL portion of Daplex. Also introduced are the primary operators of the DML portion of Daplex which is used to

manipulate the primitive objects in the DDL. The purpose of this overview is to introduce the terms and concepts of Daplex in order to understand the transformation and translation of Daplex into ABDL. (See Chapters 4 and 6.)

Informal definitions of some of the above terms follow. They will be formally defined later in this section. The *primitive objects* are databases, entity types, and nonentity types. The *properties of objects* are the attributes and constraints. Intuitively, constraints specify legal values for the primitive objects. The *relations among the objects* are the generalization hierarchies. A hierarchy is a group of things that are in tree-like order from the base of the tree, called the root, to the leaves of the tree, called terminals. An individual thing in a tree is called a node. In between and inclusive of the root and the terminals may be nodes classified in relationship to their location in the tree as ancestors, descendants, types, subtypes, supertypes, parents or children. Informally, generalization hierarchies in Daplex reflect isa relationships where some entity types are subtypes of a more general entity type. For example, in the Person Generalization Hierarchy for the University Database Schema, the entity set "graduate" isa entity set "student" where "student" is the more general entity type. For a formal definition on hierarchies see Weishar's thesis [Ref. 3]. The *primary operators* which manipulate the primitive objects are the FOR EACH statement, the ASSIGNMENT statement, the CREATE statement, the INCLUDE statement, the EXCLUDE statement, the DESTROY statement, the MOVE statement, and the PROCEDURE_CALL statement.

1. The DDL Portion Of Daplex

Of Daplex, the data definition language (DDL) portion consists of a database schema of a related collection of the primitive objects, the properties of the objects, and the relations among the objects. An example of a database is the university database in Appendix A. The basic format of a database schema is as follows:

```
DATABASE db_name IS
  [ nonentity_type_declarations ]
  entity_type_declarations
  [ entity_type_constraints ]
END [ db_name ];
```

where:
db_name is the unique name of the database

optional nonentity_type_declarations are string types, scalar types, and numeric constants

entity_type_declarations are the declarations of entity types, their attributes, and the generalization hierarchies.

optional entity_type_constraints are the properties of the declared entity types that must remain invariant under any operations on values of those types.

Throughout this thesis square brackets ([,]) denote optional declarations. The database schema may be intermixed in any order. However, all types must be completely or partially declared before the name may appear in another declaration.

Examples of entity types are also provided in Appendix A in the university database example. The basic formats of an entity type are as follows:

```
(1) TYPE entity_type_name IS
      ENTITY
        [attribute_name_1 : attribute_type;
         attribute_name_2 : attribute_type;
                    .
                    .
                    .
         attribute_name_n : attribute_type;]
      END ENTITY;
```

(2) TYPE entity_type_name;

where:
entity_type_name is a unique name in the database

attribute_names are lists of one or more unique properties of an entity; if more than one name, then each attribute shares the same type

attribute_types may be strings,
scalars (integer, floating-point,
or enumeration type), entities,
nonentities or sets of any of
the above types

The second entity-type definition is a partial entity-type declaration. Partial declarations make entity names available as attribute types. An attribute type must be declared, completely or partially, before it can be referenced. A reference of entity types in this manner provides the operational link in implementing the user-defined relationships of the e-r model in Daplex. Whenever a partial declaration appears in an entity-type declaration. the complete declaration of that entity type must appear later in the same database declaration.

The relations among the objects in Daplex are reflected in the generalization hierarchies. All types in the generalization hierarchy are entity types or entity subtypes. An example of the Person generalization hierarchy is presented in Appendix A. The example shows that the entity type, Person, forms the root of a tree of built-in relationships. The nodes of the tree are entity subtypes. The names of subtypes must be unique. *Subtypes* are descendants of the root type or other subtypes. As the hierarchy is traversed downward from level to level, each subtype inherits all of the attributes of its supertypes. *Supertypes* are ancestors of subtypes. The highest level ancestor is the root type. The general formats of a subtype are as follows.

(1) SUBTYPE subtype_name IS supertype_names
ENTITY
[attribute_names_1 : attribute_type;
attribute_names_2 : attribute_type;
.
.
.
attribute_names_n : attribute_type;]
END ENTITY;

(2) SUBTYPE subtype_name;

Supertype_names is a list of one or more names of entity types and

subtypes of built-in relationships completely declared previously in an entity type declaration. Attribute names and types are the same as for entity types. The two subtype formats also correspond directly to the entity type formats. One important point to remember is that the complete declaration of subtype creates a built-in relationship.

Nonentity types describe the primitive objects that declare data types other than entities. There are three nonentity types: the base type, the subtype and the derived type. Their formats are as follows.

base type: TYPE type_name IS type_definition;

subtype: SUBTYPE subtype_name IS prev_name;

derived type: TYPE name IS NEW prev_name

The *type_definition* declares the data types of the nonentities. They may be user-defined or predefined. The predefined types are STRING, INTEGER, FLOAT, and BOOLEAN. User_defined types are strings, scalars(integer, floating-point, enumeration, and boolean), and numeric constants.

There are two types of constraints on entity types in Daplex. They are the overlap constraint and uniqueness constraint. An *overlap constraint* determines when an entity may legally belong to more than one terminal entity subtype. A terminal subtype is a leaf in the generalization hierarchy. Terminal types are disjoint unless they are connected with the overlap constraint. The general format of the overlap constraint is as follows.

OVERLAP entity_type_names WITH entity_type_names;

Entity_type_names are unique terminal subtypes. An example of the overlap constraint is in the university database example in Appendix A.

The uniqueness constraint specifies, for a particular entity type or subtype, a collection of attributes whose values are unique for all entities in a database belonging to that type or subtype. Several examples are provided in the university database example in Appendix A. The general format of the uniqueness constraint is as follows.

UNIQUE attribute_name WITHIN entity_type_name;

A restriction for values of attributes is that uniqueness constraints only apply to values directly. Attributes that derive their values from relationships with entities or nonentities are precluded from forming uniqueness constraints.

The Daplex DDL is to be discussed again in chapter 4 where transformations between Daplex and MLDS structures are presented.

2. The DML Portion Of Daplex

The data manipulation language (DML) of Daplex consists of many statements. We discuss each of the statements in this section. The FOR EACH statement is used to retrieve data from the database. The ASSIGNMENT statement is used to modify attribute values in the database. The CREATE statement is used to insert new data into the database. The INCLUDE statement is to permit additions of attribute values to set valued functions (attributes). The EXCLUDE statement is used to remove an attribute value or group of attribute values from a set valued function. The DESTROY statement is used to remove data from the database. The MOVE statement moves a record or a group of records with the same type conditions from one set of subtype files into another set of subtype files. The PROCEDURE_CALL statement allows the user to format output from transaction requests. The exact syntax and semantics of each of the primary operations is thoroughly examined in Chapter 6 when the translation process from Daplex to ABDL is explained.

An important point is to recognize that, in using the primitive operations, parenthetical functional representations of the primitive objects are used. Some examples are as follows.

name(s)
gpa(u)

The attributes, name and gpa, are examples from the university database. The variables, s and u, apply to the entities, student and undergraduate, respectively. Notice that the value of the attribute, name, in the entity set Person is given to the entity set, Student, while the value of the attribute, gpa, in the entity set, Undergraduate, is self-contained. The functional composition is applied to values of attributes from user-defined relationships. The following example from the university database illustrates functional composition.

name(advisor(s))

As before, s represents the student entity. However, the attribute, advisor, in the entity set, Student, receives its value through its user-defined relationship with the faculty entity. The entity set, Faculty, in turn, inherits its value for the attribute, name, from its built-in relationship with its supertypes employee and person. The built-in relationships are illustrated in the person generalization hierarchy schema in Appendix A. The user-defined relationships are illustrated in the graphical representation of the schema, also in Appendix A.

IV. DATA STRUCTURES NECESSARY TO EXECUTE DAPLEX

To translate Daplex transactions to ABDL, Daplex data structures must also be transformed into ABDL data structures. Once Daplex transactions are received they may be translated into their corresponding ABDL equivalent constructs. The ABDL constructs have been presented in Chapter 2, Section A. Here we focus on the data structure transformation and transaction translation.

In transforming data structures from Daplex to ABDL, the attribute-based (a-b) record structure has been created with an additional attribute-value (a-v) pair. Justification for this addition is presented in Chapter 5 on transforming the Daplex schema into a corresponding ABDL schema.

The following discussion on data structures describes the methodology used to arrive at the final design of the data structures and the design and methodology of data structures for Daplex.

A. A METHODOLOGY FOR THE DESIGN OF DATA STRUCTURES

The method for designing Daplex data structures has been provided by intuition gained through the use of the declaration section from CCA's sample university database schema and its graphical representation in their users manual. As presented in Chapter 3, a *database* is a collection of related data usually of several different types. The related data types, also presented in Chapter 3, are described in terms of relationships and entities. The database *schema* is the data definition of the types for relationships and entities. The physical representation of the schema in computer memory is via the data structure. The sample university database and the graphical representation are reproduced in Appendix A.

The goal is to apply MacLennan's abstraction principle [Ref. 21]. That is to identify and abstract useful, frequently recurring patterns of data into data structures that would represent the database schema. In representing the database it is necessary to provide entries in the Daplex data structures that

consider the primitive objects, the properties of the objects, and the relations among the objects. The data structure that applies most closely to the Daplex schema is the list. Thus, in the design of Daplex structures the list data structure has been applied across all of Daplex's primitive objects, properties of the objects, and relations among the objects in the Daplex schema.

The university database has, also, helped to provide an example of an overall picture of how relationships between entities are structured within a Daplex database or an e-r model. This example is depicted in Figure 4.1. This depiction presents the set-theoretic concepts that support Daplex and the e-r model. Daplex and the e-r model rely heavily on the set theory in their database design.

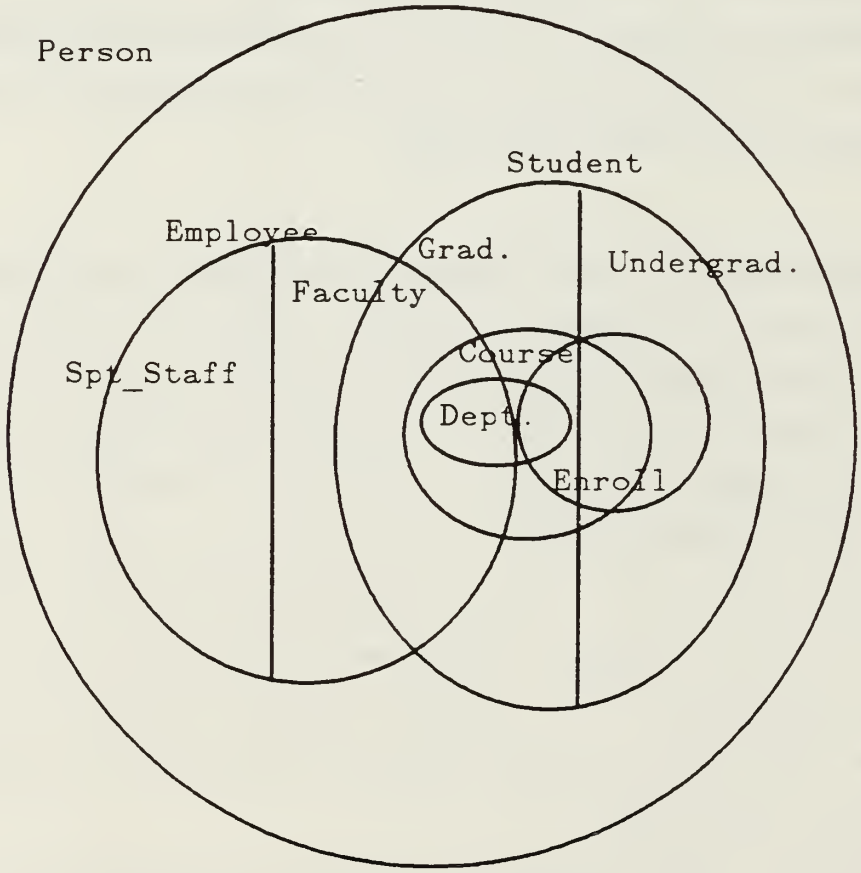


Figure 4.1 Set-Theoretic Depiction of Sample Daplex Database

B. DESIGN OF DATA STRUCTURES FOR DAPLEX

In the construction of the Daplex data structures, presented in Appendix B, it has finally been resolved, through trial-and-error, to provide a top-level database node that references or points to a list of all possible types, subtypes, and derived types of entities and nonentities. This top-level node is called the ent_dbid_node. Each type has its own list in the database.

Entity types form lists of entity types, using the ent_node structure, with individual sub-lists, using the function_node structure, of their attributes and constraints. Subtypes, using the gen_sub_node structure, form lists of entity subtypes, sub-lists of their individual supertypes (entity types or subtypes), sub-lists of their individual overlap types, using either the overlap_sub_node or the overlap_ent_node structures, if applicable, and individual sub-lists of their attributes and constraints, again using the function_node structure. Attributes of supertypes are inherited by subtypes. The inheritance is provided by the list structure from subtype to supertype.

Attributes may be completely specified or related to other entities or nonentities with single or multiple values of a single type. To allow for different attribute data types and the potential for multiple values, a sub-list structure is also created to point to the value or values of each attribute. This sub-list structure is the function_node to be described below. To allow for one or more values of one attribute, the attribute value is provided by another sub-list using the ent_value structure.

Each nonentity creates separate lists of types, subtypes, and derived types. In one of the first data structure designs, the structures for subtypes and derived types have created sub-lists of their supertypes in order, back to the root. The sub-list to point back to the root has been eliminated due to the complexity involved in the implementation of circular pointers. Since the values of nonentity types are completely specified, only lists of values needed to be maintained, instead of the list of function_nodes required by entity types. Thus a separate value node is created to list single or multiple values of these types. The ent_value structure can be used directly from the nonentity types to store the value(s) of nonentity types. This contrasts to the indirect method of storing

entity values. As a by-product of designing the nonentity type structure, it is discovered that this structure also worked for the constant declaration. The constant declaration only requires storage for its name and value, both of which are provided in the ent_non_node structure.

To handle completely specified values and relationship values in the entity types, a separate construct, called a function_node, is developed. This construct provides for all the possible values, relationships, and constraints that a function could possibly have in an entity. Note, that only one structure in the function_node is to be used to define an attribute value at any one time. Multiple values of attributes are also provided by using sub-lists of the ent_value nodes or sub-lists of a specific relationship type.

The final Daplex data structures are presented in Appendix B.

V. THE TRANSFORMATION OF DAPLEX SCHEMA INTO ABDL SCHEMA

This chapter provides the specifications for transforming the Daplex Schema into its corresponding ABDL schema. In the first section of this chapter we present the methodology and background which led us to this specification. Finally, we present the specification.

A. THE BACKGROUND AND METHODOLOGY

The method we have used to derive our specification for the transformation of Daplex schema into ABDL schema is first to compare our knowledge about the Daplex and ABDL constructs. We then construct all the ways we fit or map the Daplex constructs into the ABDL constructs. The following discussion presents our findings. In section B we present our final specification.

Given the a-v pairs in a record in ABDL, it is immediately obvious that the functions of Daplex correspond directly to the attributes of ABDL. However, there are difficulties. First, values in the a-v pair can only be single-valued whereas Daplex permits single- and multiple-valued attributes or sets of values. Second, ABDL has no direct way of representing relationships between individual records.

The first problem can be resolved by repeating records for multiple-valued attributes. The number of records required is the Cartesian product of the number of multiple-valued attributes, if only individual records are involved, i.e., relationships between records are not involved. (Storage of many records is not a problem since, in MBDS, each backend has sufficient storage.)

In one mapping relationships are resolved by just repeating all the related attributes for each related record. The repeating of related attributes provides for the unique requirement of relationships in the e-r model, and is simple to implement. As noted, the storage capacity of MBDS is capable of handling repetition.

The problem with this design becomes apparent when applying it to a real example. It is shown that records with multiple values multiply the number of

records. Records with relationships containing multiple values grow exponentially according to the depth they appear in the relationship tree in which their actual values appear. The mapping which provides the best solution to the two difficulties above is presented next in the specification.

B. THE SPECIFICATION

In this solution repeating records for multiple-valued attributes is retained. Relationships between records of different files are transformed using Chen's solution of defining an artificial attribute and its value so that a unique mapping is possible. This has been described in Chapter 3. The artificial attribute we have defined is a unique key for each entity in its corresponding entity set. For example, in the entity set Person, we have defined the attribute "Person.key," which has as its value a unique number for each entity. Thus, a unique a-v pair or keyword, consisting of the entity key and its value, is provided for each entity type or subtype. The value of this unique keyword is the "link" between entities. types or subtypes, in a relationship, i.e., entities are related only in accordance with the unique key. To retrieve attribute values in an entity type or subtype, dependent on a relationship, we reference the corresponding unique keyword directly.

As a consequent of this solution an algorithm to transform the Daplex schema into its corresponding ABDL schema is provided as follows:

(1) Separate the entity types and subtypes from the nonentity types, subtypes, and derived types in the Daplex schema.

(2) For each entity type or subtype create an ABDL file for the name of each entity type or subtype. In the first a-v pair the attribute is spelled "File" and the value is the name of the entity type or subtype from the declaration or data definition section of the database schema.

(3) For each ABDL file that is an entity type, add the unique keyword as the second a-v pair of each record. The attribute consists of the file name followed by a dot and completed with the attribute, key. The value is a distinct number for each record in the file.

(3a) For each ABDL file that is an entity subtype add the unique keyword as the second a-v pair of each record. The attribute consists of file name followed by a dot followed with the attribute, key, followed by a left parenthesis, recursively followed by the related entity subtype file's (if any) keyword attribute and left parenthesis, until the related entity type is reached. At this time the keyword for the entity type is added followed by the number of right parenthesis required to balance the number of left parenthesis in the entity subtype keyword.

(4) For each attribute (function) in an entity, add an a-v pair to its corresponding record in its ABDL file. The attribute of the a-v pair is the name of the attribute (or function). The value of the a-v pair is, of course, its value.

(4a) For multiple values or set valued functions, multiple records need be created to account for all the values of an attribute. This requires the duplication of all a-v pairs of that record and the insertion of the next value of a set valued function. Attributes with set valued functions are initialized as null to prevent unwanted record deletion for empty sets.

(4b) For attribute values that derive their values from relationships with entities, the attribute of the a-v pair consists of the attribute name, followed by a left parenthesis, recursively followed by a related entity subtype file's (if any) keyword attribute and left parenthesis, until the related entity type is reached. At this time the keyword for the entity type is added followed by the number of right parenthesis required to balance the number of left parenthesis in the entity subtype keyword.

(4c) For single valued attributes, with no relationship dependencies, insert its value in the second position of the a-v pair.

Applying the above algorithm to the sample Daplex University database schema in Appendix A, we transform the Daplex University database schema into its corresponding ABDL schema. This transformation is depicted by the templates of the ABDL University database schema in Figure 5.1. Asterisks in the templates represent values dependent upon relationships. Other values in the templates represent the entity or the attribute type. Nonentity types are treated as user-defined types. *Templates* are general representations of an actual or fully specified database schema. Templates represent the schema without showing the fully specified representation of records for set-valued functions. The translation

of the CREATE statement in Chapter 6 shows an example between templates and a fully specified representation.

```
(<File, person>, <person.key, **>, <name, string>,
 <ssn, string = 000000000>)

(<File, employee>, <employee.key, <person.key, **>>,
 <home_address, string>, <office, string>,
 <phones, set of string>, <salary, float>,
 <dependents, integer range>)

(<File, support_staff>,
 <support_staff.key, <employee.key, <person.key, **>>>,
 <supervisor, <employee.key, <person.key, **>>>,
 <full_time, boolean>)

(<File, faculty>,
 <faculty.key, <employee.key, <person.key, **>>>,
 <rank, rank_name>, <teaching, <course.key, ***>>,
 <tenure, boolean = FALSE>,
 <dept, <department.key, ****>>)

(<File, student>,
 <student.key, <person.key, **>>,
 <advisor,<faculty.key,<employee.key,<person.key,**>>>>,
 <major, <department.key, ****>>,
 <enrollments, <set of enrollment.key, *****>>)

(<File, graduate>,
 <graduate.key, <student.key, <person.key, **>>>,
 <advisory_committee,<faculty.key, <employee.key,
 <person.key,**>>>>)

(<File, undergraduate>,
 <undergraduate.key, <student.key, <person.key, **>>>,
 <gpa, grade_point>, <year, integer range 1 .. 4 := 1>)

(<File, course>,
 <course.key, ***>, <title, string>
 <dept, <department.key, ****>,
 <semester, semester_name>, <credits, integer>)

(<File, department>, <department.key, ****>,
 <head,<faculty.key,<employee.key,<person.key,**>>>>,
```

(<File, enrollment>, <enrollment.key, *****>,
 <class, <course.key, ***>>,
 <grade, grade_point>)

Figure 5.1 Template Of ABDL University Database Schema

VI. TRANSLATING DAPLEX TRANSACTIONS TO ABDL TRANSACTIONS

This section describes the translation of Daplex transactions to their equivalent ABDL transactions. The previous design theses, referenced in Chapter 1, referred to the translation process as a mapping operation. Translation and mapping are synonomous in the context of these theses. The Daplex transactions to be described involve the primary operators for looping and updating. The Daplex looping operator is the FOR EACH loop. The Daplex updating operators are the ASSIGNMENT, INCLUDE, EXCLUDE, CREATE, DESTROY, MOVE, and PROCEDURE_CALL statements. Each of these expressions and their associated ABDL mappings is discussed below.

ABDL statements, as stated in Chapter 2, are RETRIEVE, RETRIEVE-COMMON, INSERT, UPDATE, and DELETE. The manipulation of these ABDL statements for the purpose of mapping the Daplex statements becomes apparent in the examples below. Once again examples are used to gain the necessary intuition in order to formulate a general mapping for each Daplex expression. The examples are taken from CCA's Daplex User's Manual [Ref. 18]. The CREATE statement is selected first because it seems natural to start with the creation of a database; and, to provide clues about similarities between the ABDL INSERT and the Daplex CREATE.

Neither Daplex's PROCEDURE_CALL statement nor ABDL's RETRIEVE-COMMON statement will be presented in this chapter. The PROCEDURE_CALL which includes procedures like print and cancel are accommodated by the MLDS and ABDL operators. The RETRIEVE-COMMON is not used in any of the Daplex examples.

A. THE CREATE STATEMENT

The CREATE statement is used to insert new data into the database. More specifically, the CREATE statement creates a new database entity. The general case of CREATE (Figure 6.1) consists of one or more previously declared entity types or subtypes. Just as in imperative programming languages like Pascal, specific data structures must be declared in the declaration section defining the specific database. If multiple types are listed (entity types or subtypes), each type must be a terminal type.

The Daplex example in Figure 6.2 shows the creation of graduate and faculty subtypes. Both are terminal types, i.e., leafs in the relationship tree or the Generalization Hierarchy of Person in Appendix A. Also, shown, to the left of their corresponding functions, are nonterminal supertypes (types and subtypes in the relationship tree, e.g., person, employee, and student) of graduate and faculty. All functions receive values. They may be user-defined or default values. The function tenure in the entity faculty receives a default value. Functions with default values do not have to be specified. The system automatically assigns their values by looking them up in their declarations. It is possible to have all default values and thus no function names. Ancestors, such as these, require indirect creation whenever their descendents, (i.e., faculty and graduate), are created. Faculty and graduate are declared as overlapping entities (see Chapter 3). Thus, they have the same attribute characteristics of person.

The equivalent ABDL example in Figures 6.3a and 6.3b is a sequence of

```
CREATE NEW entity_type_names
  [(function_name_1 => expression_1,
    function_name_2 => expression_2,
                    .
                    .
                    .
    function_name_n => expression_n)];
```

Figure 6.1 The CREATE in Daplex

```
CREATE NEW graduate, faculty
        -- (name          => "Jane Jones",
person  |__ssn           => "22331111",
        -- home_address   => "503 S. Atherton",
        |  office         => "104",
employ  |  phones         => {"4928860", "5327020"},
        |  salary         => 5000.00,
        |__dependents     => 0,
        -- rank           => assistant,
        |  teaching       =>
        | {c IN course WHERE title(c) = {"CS2970","CS3111"}},
fac'lt  |   (tenure  - default )
        |   dept          =>
        |__{d IN department WHERE dname(d) = "CS"},
        --  advisor       =>
        | {f IN faculty WHERE name(f) = "MacLennan"},
stud't  |   major         =>
        | {d IN dept WHERE dname(d) = "CS"},
        |   enrollments   =>
        | {e IN enrollment WHERE title(e) =
        __{"CS3450", "CS4112", "CS4150"}},
grad    |advisory_committee =>
        | {f IN faculty WHERE name(f) = {"Hsiao", "Davis"}});
```

Figure 6.2 A Daplex Sample for the CREATE

non-hierarchical, fixed-length record types. These record types may be entity types or subtypes in accordance with the daplex schema. The thirteen statements in Figure 6.3a show the ABDL templates that correspond to the Daplex CREATE. These templates are the actual ABDL instructions to implement the CREATE. The disjunction in the ABDL RETRIEVE operation repeats the INSERT instruction until there are no more lookup functions to be retrieved. The nineteen statements in Figure 6.3b show the actual effect of the ABDL templates on the database. For example, it is shown that three separate records are created for the student entity and the enrollment entity for each course in which the student enrolled. Later statement mappings do not reflect the effect of ABDL templates. However, the reader should be cognizant of the

effect. Spaces are left between entities to reflect the corresponding mappings for each created entity.

With the intuition gained from the above example, the general case of translating the CREATE statement in a sequence of ABDL operations can be formulated. As shown in Figure 6.4, the translation consists of a sequence of INSERT and RETRIEVE operations. The sequence is dependent upon the actual expression value, i.e., the value on the right hand side of the imply sign (=>) in Figure 6.1. If the expression value is a single actual value, then only one INSERT is generated. If the expression value is a set of actual values, then multiple INSERTs are generated for multiple values. This is reflected by the function phones in the entity employee. Otherwise, expression obtains its value from relationships between entities. There are several instances of this in the example. One is reflected in the function enrollments in the entity student. In that case, a RETRIEVE, in disjunctive normal form, is used to return a key to the value of the expression. This is shown in the example in Figure 6.3b.

Before beginning a CREATE the relationship tree must be traversed through all its ancestors to its ultimate ancestor, the root. Each ancestor, in addition to the file specified in the CREATE, must be created with the appropriate function values as described in the example. The ancestors and the specified file(s) then determine the number of INSERTs required for that CREATE transaction. The order that the functions occur in the CREATE transaction follows the prefix structure of their corresponding entity(s) in the relationship tree up until the terminal entity(s) specified.

The general case of the corresponding ABDL CREATE in Figure C.1 of Appendix C is a mapping algorithm, not an execution algorithm. A *mapping algorithm* is a method for specifying all possible translations from a Daplex transaction to its equivalent transaction in ABDL. An *execution algorithm* is a method for implementing the specifications produced by the mapping algorithm so that the equivalent ABDL transaction may be run. Figures 6.3a and 6.3b are examples of an execution algorithm.

After the file(s) or entity(s) have been located in the Daplex CREATE statement, the mapping algorithm for the CREATE evaluates the expressions for

```
1. INSERT(<File, person>,
        <person.key, next_file_seq_numb= *>,
        <name, Jane Jones>, <ssn, 111223333>)

  /* Following two INSERTS on employee because phones
     are completely specified */
2. INSERT(<File, employee>, <employee.key,<person.key, *>>,
        <home_address, 503 S. Atherton>,
        <office, 104>, <phones, 4928860>,
        <salary, 5000.0>,
        <dependents, 0>)
3. INSERT(<File, employee>, <employee.key,<person.key, *>>,
        <home_address, 503 S. Atherton>,
        <office, 104>, <phones, 5327020>,
        <salary, 5000.0>,
        <dependents, 0>)

4. RETRIEVE(<File=course> and <title=CS2970> or
          <File=course> and <title=CS3111>) (course.key)
5. RETRIEVE (<File=department> and <dname=CS>)
          (department.key)
6. INSERT(<File, faculty>,
        <faculty.key,<employee.key,<person.key, *>>>,
        <rank, assistant>,
        <teaching,<course.key, **>>,
        <dept,<department.key,***>>)

7. RETRIEVE(<File=person> and <name=MacLennan>)(person.key)
8. RETRIEVE(<File=department> and <dname=CS>)
          (department.key)
9. RETRIEVE(<File=course> and <title=CS3450> or
           <File=course> and <title=CS4112> or
           <File=course> and <title=CS4150>) (course.key)
10. INSERT(<File,enrollment>,<enrollment.key,****>,
        <class,<course.key,**>>,<grade,0.0>)
11. INSERT(<File,student>, <student.key,<person.key,**>>,
        <advisor,<faculty.key,
          <employee.key,<person.key,**>>>>,
        <major,<department.key,***>>,
        <enrollments,<enrollment.key,**>>)

12. RETRIEVE(<File=person> and <name=Hsiao>  or
           <File=person> and <name=Davis>) (person.key)
```

```
13. INSERT (<File,graduate>,
        <graduate.key,<student.key,<person.key,**>>>,
        <advisory_committee,<faculty.key,
          <employee.key,<person.key,**>>>>)
```

Figure 6.3a An Equivalent Example of CREATE in ABDL Templates

The effect of requests executed in the templates will look like:

```
1. INSERT(<File, person>,
        <person.key, next_file_seq_numb= *>,
        <name, Jane Jones>, <ssn, 111223333>)

2. INSERT(<File, employee>, <employee.key,<person.key, *>>,
        <home_address, 503 S. Atherton>,
        <office, 104>, <phones, 4928860>,
        <salary, 5000.0>,
        <dependents, 0>)
3. INSERT(<File, employee>, <employee.key,<person.key, *>>,
        <home_address, 503 S. Atherton>,
        <office, 104>, <phones, 5327020>,
        <salary, 5000.0>,
        <dependents, 0>)

4. RETRIEVE(<File=course> and <title=CS2970> or
          <File=course> and <title=CS3111>) (course.key)
5. RETRIEVE (<File=department> and <dname=CS>)
          (department.key)
6. INSERT(<File, faculty>,
        <faculty.key,<employee.key,<person.key, *>>>,
        <rank, assistant>,
        <teaching, course.key -> CS2970>,
        <tenure, false>,
        <dept, department.key -> CS>)
7. INSERT(<File, faculty>,
        <faculty.key,<employee.key,<person.key, *>>>,
        <rank, assistant>,
        <teaching, course.key -> CS3111>,
        <tenure, false>,
        <dept, department.key -> CS>)


8. RETRIEVE(<File=person> and <name=MacLennan>)(person.key)
9. RETRIEVE(<File=department> and <dname=CS>)
          (department.key)
10. RETRIEVE(<File=course> and <title=CS3450> or
           <File=course> and <title=CS4112> or
           <File=course> and <title=CS4150>) (course.key)
11. INSERT(<File,enrollment>,<enrollment.key,****>,
        <class,<course.key,**>>,<grade,0.0>)
12. INSERT(<File,student>, <student.key,<person.key,**>>,
        <advisor,<faculty.key,
```

```
        <employee.key,<person.key,**>>>>,
      <major,<department.key,***>>,
      <enrollments,<enrollment.key,**>>)
13. INSERT(<File,enrollment>,<enrollment.key,****>,
      <class,<course.key,**>>,<grade,0.0>)
14. INSERT(<File,student>, <student.key,<person.key,**>>,
      <advisor,<faculty.key,
        <employee.key,<person.key,**>>>>,
      <major,<department.key,***>>,
      <enrollments,<enrollment.key,**>>)
15. INSERT(<File,enrollment>,<enrollment.key,****>,
      <class,<course.key,**>>,<grade,0.0>)
16. INSERT(<File,student>, <student.key,<person.key,**>>,
      <advisor,<faculty.key,
        <employee.key,<person.key,**>>>>,
      <major,<department.key,***>>,
      <enrollments,<enrollment.key,**>>)

17. RETRIEVE(<File=person> and <name=Hsiao>  or
      <File=person> and <name=Davis>) (person.key)
18. INSERT (<File,graduate>,
      <graduate.key,<student.key,<person.key,**>>>.
      <advisory_committee,<faculty.key,
        <employee.key,<person.key,**>>>>)
19. INSERT (<File,graduate>,
      <graduate.key,<student.key,<person.key,**>>>,
      <advisory_committee,<faculty.key,
        <employee.key,<person.key,**>>>>)
```

Note: Multiple function values, fully specified (actual) or referenced, create multiple records.

Figure 6.3b An Equivalent Example of the CREATE in ABDL

each function of an entity. If the function expression requires a reference to a value in a relationship with another entity, the algorithm retrieves the entity(s) and returns the reference (i.e., pointer) to its value(s).

Sometimes it is necessary to create an entity just to be able to return a reference to a value. Such a case occurs with the entity type enrollment. Before the referenced value for attribute enrollments can be entered in the entity student, entity enrollment must be created for each course in which that student

is to enroll. Finally, regardless of whether the functions of an entity receive actual or referenced values from their expressions, the entity is created. When the specified file(s) or entity(s) have been created the CREATE statement is completely mapped.

Note that this mapping algorithm only specifies one error condition. There are three other types of errors possible: undeclared functions in an entity, illegal function expression types, and illegal overlapping of entities. These errors are caught by the DML routine in Chapter 7 that checks for data-structure correctness. The point here is that these three types of errors are implementation errors that do not relate to the mapping process.

B. THE DESTROY STATEMENT

The DESTROY statement is used to remove data from the database. It reverses the effect of the CREATE statement. The corresponding ABDL statements are RETRIEVE and DELETE. The statement of DESTROY is as follows:

DESTROY (entity_valued_expression);

An entity_valued_expression is an entity-valued loop parameter, a single, entity-valued function expression, or an entity-valued set containing one member. The entity (record) is deleted from all types and subtypes to which it belongs and is completely removed from the database.

An example is provided in Figure 6.4. The mapping algorithm is displayed in Figure C.2 of Appendix C.

DESTROY {f in faculty WHERE ssn(f) = "111223333"};

The following is the execution algorithm:

```
Search entity types and subtypes for
                    function_expressions for
        faculty.key
Return  file = student, file = graduate,
                      file = department;
```

```
RETRIEVE (<File = person> and <ssn = "111223333">)
    (person.key)
 for each (person.key)
if not (RETRIEVE (<File = student> and <advisor = *>)
       (student.key)) or
   not (RETRIEVE (<File = graduate> and <advisor = *>)
       (graduate.key)) or
   not (RETRIEVE (<File = department> and <head = *>)
       (department.key))
then
DELETE (<File = faculty> and
       <faculty.key<employee.key<person.key = *>>>)
DELETE (<File = employee> and
       <employee.key<person.key = *>>)
DELETE (<File = person> and <person.key = *>)
```

Figure 6.4 Example of DESTROY Statement

C. THE FOR EACH LOOP

The FOR EACH statement is a loop to retrieve data from the database. The FOR EACH loop is shown in Figure 6.5. The FOR EACH loop performs instructions for each member in a set of database values given in set_expression. The loop_label functions as a "goto" instruction to exit from the loop_body when certain conditions are satisfied.

The loop_parameter is a variable for an attribute of the set of database values. It represents the same attribute for the life of the loop_body. The set_expression provides the set of actual values for the loop_body.

In the remainder of this section examples from the Daplex manual are used to demonstrate Daplex to ABDL translations (Figure 6.6). For all of the

```
[loop_label:] FOR [EACH] loop_parameter IN set_expression
  [WHERE boolean_expression] [BY order_clause]
[LOOP]
   loop_body
END [LOOP];
```

Figure 6.5 The FOR EACH Loop

following examples, the RETRIEVE in the data manipulation language of ABDL provides the same function that the Daplex PRINT command does. The RETRIEVE executes in the following steps. First, the KC of MLDS gets a buffer for a particular Daplex function or attribute, such as person.key. It then takes the value of person.key from that buffer, which is actually a temporary file, and uses it to get the name from the person file. Finally, it prints the name. The function in the target list of the RETRIEVE, e.g., name, is the value printed.

Example C.1

```
FOR EACH s IN student
LOOP
  PRINT (name(s));
END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE (File = student)(person.key)
 for each person.key
RETRIEVE ((File = person) and (person.key = ***))
                                  (name)
```

Example C.2

```
FOR EACH s IN student BY ASCENDING name(s)
LOOP
   PRINT (name(s));
   PRINT (name(advisor(s));
END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE (File = student)(person.key)
 for each person.key
RETRIEVE ((File = person) and (person.key = **))
                 (name,person.key) BY name
 for each person.key
RETRIEVE ((File = student) and (person.key = **))
                                (advisor)
 for each person.key
RETRIEVE ((File = person) and (person.key = **))
                                  (name)
```

Example C.3

```
FOR EACH e IN enrollments(student)
  WHERE credits (class(e)) > 3
  BY semester (class(e))
LOOP
   PRINT  (title(class(e)));
END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE (File = student)(enrollments)
/* enrollments -> enrollment.key */
  for each enrollment.key
RETRIEVE ((File = enrollment) and
        (enrollment.key = **))(class)
  for each course.key
RETRIEVE ((File = course) and
        (course.key = **) and (credits > 3))
                    (title) BY semester
```

Example C.4

```
FOR EACH u IN undergraduate
  WHERE COUNT (enrollments(u)) > 4 AND
    (gpa (u) < 2.5 OR FOR SOME e in
     enrollments(u) : (grade (e) < 1.5))
LOOP
  PRINT (name(u));
END LOOP;
```

The following is the execution algorithm:

```
      /* Note: cannot include aggregate operators
      /* as one of RETRIEVE arguments.  May only
      /* include in target list. */

RETRIEVE (File = undergraduate)(person.key)
 for each person.key
RETRIEVE ((File = student) and (person.key = **))
       (enrollments,COUNT(enrollments),person.key)
  if COUNT > 4
  for each person.key
RETRIEVE ((File = undergraduate) and (gpa < 2.5))
                  (person.key) BY (person.key)
      /* BY sorts by person.key and
      /*  eliminates duplicates */
```

```
  for each enrollment.key
RETRIEVE ((File = enrollments)
        and (enrollment.key = **)
        and ( grade < 1.5))(enrollment.key)
  for each enrollment.key
RETRIEVE ((File=student) and (enrollment.key=**))
                  (person.key) BY (person.key)
/* Check and discard duplicates from student
/* retrieve.
/* Then compare both person buffers and eliminate
/* duplicates by merging into one new person file.
/* Finally retrieve names from person file. */

RETRIEVE ((File = person) and (person.key = **))
                                  (name)

  /* For the general case will have to specify
  /* two or more pointers to buffers of the same
  /* named target list.  Number of pointers will
  /* depend upon number of "OR's".
```

Example C.5

```
  FOR EACH s IN student BY name(s)
  LOOP
    PRINT (name(s));
    FOR EACH e IN enrollments(s)
    WHERE credits (class(e)) > 3
  /* where's implemented as arguments except after
  /* "OR's". */
    BY semester (class(e))
    DESCENDING credits(class(e))
  LOOP
    PRINT (title(class(e)));
  END LOOP;
  END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE (File = student) (person.key)
  for each person.key
RETRIEVE ((File = person) and (person.key = *))
                  (name.person.key) BY name
  for each person.key
RETRIEVE ((File = student) and (person.key = *))
                          (enrollments)
```

```
 for each enrollment.key
RETRIEVE ((File = enrollments) and
        (enrollment.key = **)) (class)
  for each course.key
RETRIEVE ((File = course) and (course.key = *)
         and (credits > 3))
        (title) BY (semester, INVERT(credits))
```

```
Example C.6
FOR EACH c IN course
LOOP
  FOR EACH d IN department
    WHERE name(d) = name(dept(c)) and
        head(d) != NULL
  LOOP
    PRINT (name(head(d)));
    PRINT (title(c));
  END LOOP;
END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE (File = course) (course.key,dept)
  for each course.key
  for each department.key
RETRIEVE ((File = dept) and (department.key = **)
                    and (head != 0))
                   (department.key,head)
  for each person.key
RETRIEVE ((File = person) and (person.key = **))
                                    (name)
  for each department.key
RETRIEVE ((File = course) and (dept = department.key)
                                      (title)
```

```
Example C.7
FOR EACH s IN student WHERE advisor(s) != NULL
LOOP
  FOR EACH c IN enrollments(s)
  LOOP
    PRINT (name(s));
    PRINT (title(class(c)));
  END LOOP
  FOR EACH t IN teaching (advisor(s))
```

```
  LOOP
    PRINT (name(advisor(s)), title(t));
  END LOOP;
END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE ((File = student) and (advisor != 0))
                    (person.key, advisor)
  for each person.key
RETRIEVE ((File = person) and (person.key = **))
                                  (name)
RETRIEVE ((File = student) and (person.key = **))
                               (enrollments)
RETRIEVE ((File = enrollments) and
          (enrollment.key = **)) (class)
  for each course.key
RETRIEVE ((File = course) and (course.key = **))(title)
RETRIEVE ((File = person) and (advisor = **))(name)
RETRIEVE ((File = faculty) and
        (advisor(person.key) = **)) (teaching)
  for each course.key
RETRIEVE ((File = course) and (course.key = **)) (title)
```

Figure 6.6 Examples Of FOR EACH Loop And Their ABDL Translations

A mapping algorithm is presented in Figure C.3 of Appendix C for the FOR EACH loop. For any loop_parameter, such as in Example C.3 where e represents the enrollments, the algorithm first retrieves a set of all possible database values. Then, on the basis of values received and the function declared (in the loop_body or in the set_expression), the algorithm proceeds to translate the instructions in the loop_body. The loop_body instructions are represented recursively inside the square brackets in the mapping algorithm.

ABDL automatically returns values in ascending order. The order_clause DESCENDING in Example C.5 is not necessary in the mapping algorithm. The KC in the language interface will translate DESCENDING by returning values in reverse order. The "BY" instruction in ABDL is the same as in Daplex.

D. THE ASSIGNMENT STATEMENT

The ASSIGNMENT statement is used to modify function values in the database. The ASSIGNMENT statement is very close to the LOOP examples. In fact, except for using the assignment statement in the body of the LOOP. their forms are identical. The assignment statement provides a capability to assign or change a value of a single_valued_function_expression. A *single_valued_function_expression* is an attribute (function) with only one distinct value, i.e., it is not a set-valued function. For example, if an employee's salary is increased, ASSIGN (:=) will permit the change. Similarly, if a student's gpa needs to be changed, ASSIGN makes the change.

The format for an ASSIGN is as follows:

```
[loop_label:] FOR [EACH] loop_parameter
              IN set_expression
  [WHERE boolean_expression] [BY order_clauses]
LOOP
  single_valued_function_expression := expression;
                        .                    ;
                        .                    ;
                        .                    ;
END LOOP;
```

Examples with comments are provided in Figure 6.7. The mapping algorithm is shown in Figure C.4 of Appendix C.

Example D.1

```
FOR EACH s IN student
        WHERE name(s) = "Thomas Jefferson"
LOOP
  gpa(s) := 3.7;
END LOOP;
```

The following is the execution algorithm:

```
        /* example of descendant */

RETRIEVE ((File = person) and
       (name = "Thomas Jefferson")) (person.key)
 for each (person.key)
      /* find function(s) in appropriate file */
UPDATE ((File = undergraduate) and
```

```
  (undergraduate.key(student.key(person.key = *))))
                              (gpa = 3.7)
```

Example D.2

```
FOR EACH u IN undergraduate WHERE name(s) = "cow"
LOOP
 advisor(u) :=
    {f IN faculty WHERE name(f) = "Mary Jones"};
END LOOP;
```

The following is the execution algorithm:

```
       /* example of ancestor */

RETRIEVE ((FILE = person) and (name = "cow"))
                              (person.key)
 for each (person.key = *)
RETRIEVE (File = undergraduate) and
   (undergraduate.key(student.key(person.key = *))))
       (undergraduate.key(student.key(person.key)))
     /* find faculty member */
RETRIEVE ((File = person) and (name = "Mary Jones"))
                                 (person.key)
 for each (person.key = **)))
RETRIEVE ((File = faculty) and
      (faculty.key(employee.key(person.key = **))))
         (faculty.key(employee.key(person.key)))
    /* find update function(s) in appropriate file */
UPDATE ((File = student) and
      (student.key(person.key = *)) (advisor = **))
```

Example D.3

```
 /* if following were in above loop error would occur
    since function was not declared with null */

 major(u) := NULL;
```

Example D.4

```
 gpa(u) := avg (grade DUPLICATES (enrollments(u)));

 find enrollments in file = student
```

```
go to file = enrollment and for all grades for which
u is enrolled get the average using aggregate
operation.
```

Example D.5

```
FOR EACH d IN department WHERE dname(d) = "CS"
LOOP
  head(d) :=
    {u IN undergraduate WHERE name(u) = "moose"};
END LOOP;
```

```
This assignment will cause an error, since the head
function is declared as a faculty type and the
undergraduate type does not
overlap with the faculty type.
```

Figure 6.7 Examples of The ASSIGNMENT Statement

E. THE INCLUDE STATEMENT

The INCLUDE statement is used to add attribute values to set-valued functions (attributes). As shown in the explanatory examples, each of the INCLUDE statements requires the FOR EACH loop to first find the file. The corresponding ABDL statements are a sequence of RETRIEVE operations (for the FOR EACH) followed by one or more INSERT operations (for the INCLUDE).

The format of the INCLUDE statement is:

```
[loop_label:] FOR [EACH] loop_parameter
          IN set_expression
  [WHERE boolean_expression] [BY order_clauses]
LOOP
  INCLUDE expression
        INTO set_valued_function_expression;
            .                          ;
            .                          ;
            .                          ;
END LOOP;
```

An expression is any string, scalar or entity-valued expression that yields a single

value or a set of values. A set_valued_function_expression is any function expression where the function_name has been declared to be set_valued and the function argument is a single_valued_entity_expression, an entity_valued_loop_parameter or a single_entity_valued_function_expression. The type of the expression values must be the same as the type of the function name. An error also occurs if any value falls outside of the declared legal range of values for the function.

Examples with comments are provided in Figure 6.8. The general mapping algorithm is shown in Figure C.5 of Appendix C.

Example E.1

```
FOR EACH s IN student WHERE name(s) = "Thomas Jefferson"
LOOP
  INCLUDE { c IN course WHERE title(c) = "History 1"}
    INTO enrollments(s);
END LOOP;
```

The expression { c IN course WHERE title(c) = "History 1"} is an example of an entity-valued expression that defines a set of course entities that can be treated as a unit. All course entities titled "History 1" would be included in the enrollments set for each student named Thomas Jefferson. EXCLUDE is the corresponding statement used to remove values from a set.

Enrollments is an example of a set_valued_function_expression. So if s is a loop_parameter which ranges over student, this adds the course "History 1" to the set of enrollments of s. Or s could represent one student only depending upon the conditions in WHERE of FOR EACH. In the example above , if "Thomas Jefferson" is a unique name, then only s for "Thomas Jefferson" would have "History 1" added to his set of enrollments in his student record.

The following is the execution algorithm:

```
RETRIEVE (<File = person> and
        <name = "Thomas Jefferson">) (person.key)
RETRIEVE (<File = course> and <title = "History 1">)
                                (course.key)
```

```
 for each (person.key)
/* check if student - if not error determined by KC*/
RETRIEVE (<File =student> and
        <student.key<person.key = *>>) (all)
INSERT (<File,enrollment>,
       <enrollment.key,**>,<class,<course.key,**>>,
       <grade,0.0>)
  /* new enrollment.key only */
INSERT (<File,student>, <student.key<person.key,*>>,
       <advisor,***>, <major,****>,
       <enrollments,<enrollment.key,**>>)
```

Example E.2

```
FOR EACH f IN faculty WHERE name(f) = "Cow"
LOOP
 INCLUDE {c IN course WHERE title(c) = "DBMS"
        AND dname(dept(c)) = "EECS"}
    INTO teaching(f);
END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE (<File = person> and <name = "Cow">)
                          (person.key)
RETRIEVE (<File = department> and <dname = "EECS">)
                            (department.key)
RETRIEVE (<File = course> and <title = "DBMS"> and
        <dept = **>) (course.key)
 for each (person.key)
RETRIEVE (<File = faculty> and
       <faculty.key<employee.key<person.key = *>>>)
          (all)
    /* new teaching function only */
INSERT (<File,faculty>,
       <faculty.key,<employee.key,<person.key,*>>>,
       <rank_name,****>,<teaching,<course.key,***>>,
       <tenure,*****>,<dept,******>)
```

Example E.3

```
FOR EACH e IN employee WHERE name(e) = "Bug"
LOOP
 INCLUDE "4848151"
```

```
    INTO phones(e)
END LOOP
```

The following is the execution algorithm:

```
RETRIEVE (<File = person> and <name = "Bug">)
                          (person.key)
 for each (person.key)
RETRIEVE (<File = employee> and
        <employee.key<person.key = *>>) (all)
    /* new phones function only */
INSERT (<File,employee>,<employee.key<person.key,*>>,
       <home_address,XXXX>,<office,XXXX>,
       <phones,"4848151">, <salary,XXXX>,
       <dependents,XXXX>)
```

Figure 6.8 Examples of INCLUDE Statement

F. THE EXCLUDE STATEMENT

The EXCLUDE statement is the opposite of the INCLUDE statement. Its purpose is to remove a value or group of values from a set valued function. As shown in the explanatory examples, each of the EXCLUDE statements also requires the FOR EACH loop to first find the file. The corresponding ABDL statements are a sequence of RETRIEVE operations (for the FOR EACH) followed by a DELETE operation (for the EXCLUDE). The DELETE statement has the capability to undo the ABDL INSERT statement, completely or partially. Also, the DELETE may not remove anything from the database if expression, in the general format of EXCLUDE below, specifies something not in the database, i.e., an incorrect phone number to be excluded. The INSERT statement has been discussed in its use in translating the Daplex CREATE statement and the DELETE statement has been introduced in the DESTROY statement.

To permit partial deletion, records (entities) that contain set-function expressions will be initialized with the null value. This will permit exclusion of individual items from sets of values.

The general format of the EXCLUDE statement is:

EXCLUDE expression FROM set_valued_function_expression

Expression and set_valued_function_expression are as previously described in the INCLUDE statement. Examples with comments are provided in Figure 6.9. The general mapping algorithm is shown in Figure C.6 of Appendix C.

Example F.1

```
FOR EACH s IN student WHERE name(s)="Thomas Jefferson"
LOOP
  EXCLUDE {c IN course WHERE title(c) = "History 1"}
    FROM enrollments(s);
END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE (<File = person> and
          <name = "Thomas Jefferson">)
                      (person.key)
  for each (person.key)
RETRIEVE (<File = student> and
         <student.key<person.key = *>>)
                    (enrollment.key)
  for each (enrollment.key)
RETRIEVE (<File = course> and <title = "History 1">)
                       (course.key)
DELETE (<File = enrollment> and <enrollment.key = **>
      and <class<course.key = ***>>)
DELETE (<File = student> and
        <student.key<person.key = *>>
        and <enrollments<enrollment.key = **>>)
```

Example F.2

```
FOR EACH f IN faculty WHERE name(f) = "Cow"
LOOP
  EXCLUDE {c IN course WHERE title(c) = "DBMS"
          AND dname(dept(c)) = "EECS"}
     FROM teaching(f);
END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE (<File = person> and <name "Cow">)
                        (person.key)
RETRIEVE (<File = department> and <dname = "EECS">)
                           (department.key)
RETRIEVE (<File = course> and <title = "DBMS"> and
        <dept<department.key = **>>) (course.key)
 for each (person.key)
DELETE (<File = faculty> and
              <faculty.key<person.key = *>> and
              <teaching<course.key = ***>>)
```

Example F.3

```
FOR EACH e IN employee WHERE name(e) = "Bug"
LOOP
 EXCLUDE "4848151"
  FROM phones(e);
END LOOP;
```

The following is the execution algorithm

```
RETRIEVE (<File = person> and <name = "Bug">)
                          (person.key)
 for each (person.key)
DELETE (<File = employee> and
       <employee.key<person.key = *>> and
       <phones = "4848151">)
```

Example F.4

```
FOR EACH e IN employee WHERE name(e) = "Bug"
LOOP
 EXCLUDE phones(s)
   FROM phones(s);
END LOOP;
```

The following is the execution algorithm;

```
RETRIEVE (<File = person> and <name = "Bug">)
                          (person.key)
```

```
 for each (person.key)
DELETE (<File = employee> and
      <employee.key<person.key = *>> and
      <phones != 0>)
```

Figure 6.9 Examples of EXCLUDE Statement

G. THE MOVE STATEMENT

The MOVE statement moves a record (i.e., an entity) or a group of records with the same value conditions in a subtype from one set of subtype files into another set of subtype files. All corresponding descendant subtypes of the record are also automatically moved. The subtype file into which the record is to be added must be a terminal type. When the record is added to the terminal type, it is automatically added to the ancestors of this terminal type of which part of the record does not already belong.

The general format of the MOVE statement is:

```
MOVE entity_valued_expression
   [FROM entity_type_names]
   [INTO entity_type_names]
     [(function_name_1 => expression_1,
       function_name_2 => expression_2,
                  .
                  .
                  .
       function_name_m => expression_m)];
```

where entity_valued_expression is :
- an entity_valued_loop_parameter,
- a single_entity_valued_function_expression,
- or an entity_valued set containing one member.

where FROM entity_type_names are:
- a list of one or more previously declared entity subtypes of the same base type.

where INTO entity_type_names are:

- a list of one or more overlapping terminal subtypes of the same base type.

constraints: all four are parsing aborts:
- An entity cannot be removed from or added into a base type.
- If removing a subtype and then referencing a function corresponding to that subtype, the move is aborted.
- If MOVE invalidates null value constraint, uniqueness constraint, or an overlap constraint, abort MOVE.
- If MOVE conflicts with requirement that an entity of a given supertype must belong to at least one of its subtypes, abort move.

As in the previous sections a mapping algorithm is presented in Figure C.7 of Appendix C. First, the algorithm checks for all possible errors. If there are no errors, the algorithm proceeds to retrieve record subtypes until it has found the record or records with the necessary conditions for removal. It then removes these records and associated records from all descendants of this record subtype. Finally, it locates the terminal subtype to which the record should be added. It adds it to that terminal subtype and to all ancestors of that terminal subtype to which the record does not belong.

The corresponding ABDL statements needed to transform the Daplex move statement are: RETRIEVE, DELETE, and INSERT. The RETRIEVE finds the record, the DELETE removes it from its subtype and subtype descendants, and the INSERT creates the record in the terminal type and the appropriate terminal type ancestors.

Examples are provided with comments as before in Figure 6.10 to clarify the Move statement.

Example G.1

```
MOVE {g IN graduate WHERE ssn(g) = "556667777"}
  FROM student INTO faculty
    (home_address => "789 Cambridge St, Boston",
     office       => "218",
     salary       => 25000.00,          /* EMPLOYEE
     dependents   => 0,                  ***********/
```

```
    rank        => assistant,
    dept        => {d IN department WHERE /* FACULTY
                    dname(d) = "EECS"});   ********/
```

The following is the execution algorithm:

```
RETRIEVE (File = graduate)
        (graduate.key(student.key(person.key)))
  until
RETRIEVE((File = person) and (person.key = *)
         and (ssn = "556667777"))(person.key)
       /* graduate is student is person */
DELETE ((File = graduate) and
       (graduate.key(student.key(person.key = **))))
DELETE ((File = student) and
       (student.key(person.key = **)))
      /* faculty is employee is person */
INSERT (<File, employee>,
       <employee.key,<person.key, **>>,
       <home_address, "789 Cambridge St, Boston">,
       <office, "218">, <phones,0>,
       <salary, 25000.00>, <dependents, 0>)
RETRIEVE ((File = department) and (dname = "EECS"))
                          (department.key)
INSERT (<File, faculty>,
       <faculty.key,<employee.key, <person.key, **>>>,
       <rank, <rank_name, assistant>>,
       <teaching,<course.key,0>>,
       <tenure, false>, <dept,<department.key, ***>>)
```

Example G.2

```
FOR EACH u IN undergraduate
  WHERE year(u) = 4 AND gpa(u) > 2.5
LOOP
   MOVE u FROM undergraduate INTO graduate;
END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE ((File = undergraduate) and (year = 4) and
         (gpa > 2.5))
       (undergraduate.key(student.key(person.key)))
        /* undergraduate is student */
```

```
      /* graduate is student */
for each (undergraduate(student.key(person.key)))
DELETE ((File = undergraduate) and
    (undergraduate.key(student.key(person.key = *))
INSERT (<File, graduate>,
      <graduate.key,<student.key,<person.key, *>>>,
      <advisory_committee, <employee.key,
      <person.key, 0>>>)
          /* null = 0 */
```

Example G.3

```
FOR EACH u IN undergraduate
  WHERE year(u) = 4 AND gpa(u) > 2.5
LOOP
  MOVE u FROM student INTO graduate
    (major => {d IN department WHERE
            dname(d) = "FRENCH"};
END LOOP;
```

The following is the execution algorithm:

```
RETRIEVE ((File = undergraduate) and (year = 4) and
        (gpa > 2.5))
      (undergraduate.key(student.key(person.key)))

   /* to simulate FROM */
for each (undergraduate.key(student.key(person.key)))

DELETE ((File=student) and (student.key(person.key=**))
DELETE ((File=undergraduate) and
    (undergraduate.key(student.key(person.key=**))))

   /* to simulate INTO */
RETRIEVE ((File=department) and (dname="FRENCH"))
                        (department.key)
for each (graduate.key(student.key(person.key)))
INSERT (<File,student>,<student.key,<person.key,**>>,
      <advisor,faculty.key,
      <employee.key.<person.key,0>>>,
      <major,<department.key,***>>,
      <enrollments,<enrollment.key,0>>
INSERT (<File,graduate>,
      <graduate.key,<student.key,<person.key,**>>>,
```

```
      <advisory_committee,
        <faculty.key,<employee.key,<person.key,0>>>>)
```

Example G.4

```
MOVE {u IN undergraduate WHERE ssn(u) = "556667777"
   INTO faculty
     (home_address => "456 Inman St, Cambridge",
      ...);
```

illegal move statement; will abort because leaving
undergraduate record violates overlap
constraint between subtypes.

Example G.5

```
MOVE {g IN graduate WHERE ssn(g) = "556667777"}
  FROM graduate INTO faculty
    (home_address => "890 Charles St, Boston",
     ...);
```

illegal move statement; case of dangling subtype -
entity of given supertype must belong to one
of its subtypes, i.e., student must be
undergraduate or graduate.

Example G.6

```
MOVE {u IN undergraduate WHERE ssn(u) = "555667777"}
 FROM undergraduate INTO faculty, graduate
   (home_address    => "890 Charles St., Boston",
    office          => "271",
    salary          => 5500.0,
    dependents      => 0,
    rank            => assistant,
    dept            => {d IN department WHERE
                        name(d) = "EECS"});
```

The following is the execution algorithm:

```
RETRIEVE (File = undergraduate)
       (undergraduate.key(student.key(person.key)))
 until
```

```
RETRIEVE ((File = person) and (person.key = *) and
        (ssn = "555667777"))(person.key)
  /* simulate FROM */
DELETE ((File = undergraduate) and
       (undergraduate.key(student.key(person.key = *))))
  /* still have student supertype */

     /* faculty is employee is person */
INSERT (<File, employee>,<employee.key,<person.key, **>>,
       <home_address, "890 Charles St, Boston">,
       <office, "271">, <phones,0>, <salary, 5500.00>,
       <dependents, 0>)
RETRIEVE ((File = department) and (dname = "EECS"))
              (department.key)
INSERT (<File, faculty>,
       <faculty.key,<employee.key, <person.key, **>>>,
       <rank, <rank_name, assistant>>,
       <teaching,<course.key,0>>,
       <tenure, false>, <dept,<department.key, ***>>)

INSERT (<File, graduate>,
       <graduate.key,<student.key,<person.key, *>>>,
       <advisory_committee,
       <faculty.key,<employee.key,<person.key, 0>>>>)
```

Figure 6.10 Examples of the MOVE Statement

VII. IMPLEMENTATION SPECIFICATIONS

This chapter attempts to "bridge the gap" between the design specifications and the implementation specifications for transforming and controlling the execution of Daplex data structures into their equivalent ABDL data structures and translating and controlling the execution of Daplex requests into their equivalent ABDL requests. In Chapter 2 we have briefly presented the MLDS modules required to transfer and translate Daplex data definitions and transactions into equivalent ABDL data definitions and transactions. In Chapter 4 we have provided a rigorous specification for a Daplex database and structured the Daplex data in a format recognizable by LIL of MLDS. In this chapter, we first describe a proposed implementation specification for the transformation and translation of Daplex data definition schema and requests into equivalent ABDL data definition schema and requests using KMS of MLDS. We then describe a proposed implementation specification for the control of the ABDL data definitions and requests in MBDS using KC of MLDS. It should be noted that the proposals for KMS and KC in this chapter are not meant to be rigorous implementation specifications. Rather they are meant to pass our understanding of the operations and functions of the Daplex database system onward for implementation.

A. PARSING AND TRANSLATING BY KMS

This section proposes specifications for an implementation of the transformation of Chapter 5 and the translation of Chapter 6. The proposal parallels a method used in all previous implementations [Ref. 8, 9, 10]. The interested reader is referred to these References for details of the MLDS modules. As mentioned in Chapters 2, 5, and 6, KMS of MLDS is the module responsible for transforming and translating Daplex requests into their equivalent ABDL requests. The KMS functions include:

(1) parsing the request to validate the user's Daplex syntax, and

(2) translating the request into an equivalent ABDL request.

To propose an implementation specification for KMS we utilized the Adaplex grammar [Ref. 22] in the Backus-Naur Form (BNF) of CCA. The Adaplex BNF grammar is then reduced to a pure Daplex BNF grammar by eliminating all system-dependent and Ada-related goals and rules. A *grammar* is a set of goals and rules governing the syntax of a language. The Daplex BNF grammar accomplishes the first function of KMS. The second function of KMS is proposed by the interleaving pseudocode in the Daplex BNF grammar in the recursive-descent format. *Pseudocode* is short, concise, English language phrases that are structured by key words such as If-Then-Else, While-Do, and End. Key words and indentation describe the flow of control, while the English phrases describe processing actions [Ref. 1]. The recursive-descent format implies that the pseudocode is recursively defined in the descending order in which the rules are executed in the grammar in order to satisfy the goals of the grammar.

The Daplex BNF grammar is further subdivided into its DDL and DML sections to match the requirements for transforming and translating Daplex requests into their equivalent ABDL requests. The informal pseudocode is well over 2000 lines. Because of its length it was decided not to include it in this thesis. It is available upon request. Since its purpose is to provide a specification for actual implementation, which is 5 or 6 times longer than the pseudocode, it is recommended that the reader review the actual implementation of Daplex to ABDL [Ref. 11] for the final implementation decisions.

B. CONTROLLING BY KC

This section provides an overview of the control for executing ABDL requests by MBDS once KMS has performed the necessary transformations and translations of Daplex requests into their equivalent ABDL requests. KC controls the submission of the ABDL requests to MBDS for processing.

For ABDL requests that involve inserting information to create a new database, or the use of intermediate retrieval requests to insert, delete, or update information in an existing database, control is returned to LIL after MBDS

processes the transaction. Informally, *intermediate retrieval requests* are those requests which retrieve data values from data files which are used in other data files to retrieve data values that are needed to satisfy a transaction request. For ABDL requests involving final retrieval requests, KC sends the ABDL request to MBDS, receives the results back from MBDS, loads the results into a buffer, and calls KFS to format the results one buffer at a time. Informally, a *final retrieval request* is the last request needed to retrieve a data value to satisfy a transaction request. After the last buffer is processed by KFS the resulting table is displayed and control returns to LIL.

Examples of intermediate requests are provided in Section C of Chapter 6. Intermediate retrieval requests are sent to KC by KMS as request templates. A *request template* is an ABDL retrieve request with one unspecified attribute value. KC must use the results obtained from the previous ABDL retrieve request (i.e., target attribute values) and the request template to build the next ABDL request, i.e., KC substitutes the target attribute values for the unspecified attribute value in the request template. The processing of intermediate retrieval requests is managed by KC.

Just as data structures are specified for LIL in Chapter 4, data structures are specified for KC to recognize the different types of ABDL requests. A KC procedure then uses instructions for each separate ABDL request (i.e., INSERT, DELETE, UPDATE, the intermediate RETRIEVE, and the final RETRIEVE) to control the processing of the ABDL requests. References are again made to all the implementation theses for details of the required data structures and control procedures. A particular emphasis is directed to Reference 9 where the implementation of KC and the ABDL example are identical to that required for Daplex.

VIII. RESULTS AND CONCLUSIONS

As stated in Chapter 1, by using an unconventional approach to the design and implementation of a basic database system, we can have a system supporting multiple data models as if the system is a heterogeneous collection of database systems. Our unconventional approach is geared to flexibility, efficiency, and extensibility, which makes it an attractive alternative to conventional approaches. By developing multiple data language interfaces we offer users our approach without incurring any retraining costs. In adopting our system, users appear to have their same old database system, but one that works faster and has other data languages.

In this theses we have presented a methodology for supporting entity-relationship database management on an attribute-based database system. Specifically, we have provided the design and detailed data structures required to recognize Daplex requests by MLDS in Chapter 4. We have described the transformations required from Daplex data definitions in the Daplex schema into ABDL definitions in Chapter 5; and provided a proposed implementation specification for the DDL transformations in Chapter 7. We have also described the translations required from Daplex transactions into their corresponding ABDL translations in Chapter 6; and provided a proposed implementation specification for the DML translation in Chapter 7. Finally, we provided an overview of the control procedures required for processing the equivalent ABDL requests in Chapter 7.

The entity-relationship interface can be implemented on the basis of the work we have presented herein and of the work we have accomplished to date [Ref. 8, 9, 10]. The implementation of the entity-relationship interface using MLDS will be the high point in the study of the Multi-Backend and Multi-lingual Database System.

```
DATABASE university IS

  TYPE person;
  SUBTYPE employee;
  SUBTYPE support_staff;
  SUBTYPE faculty;
  SUBTYPE student;
  SUBYPE graduate;
  SUBTYPE undergraduate;
  TYPE course;
  TYPE department;
  TYPE enrollment;
  TYPE rank_name IS (assistant, associate, full);
  TYPE semester_name IS (fall, spring, summer);
  TYPE grade_point IS FLOAT RANGE 0.0 .. 4.0;

  TYPE person IS
    ENTITY
      name : STRING (1 .. 25);
      ssn  : STRING (1 .. 9) := "000000000";
    END ENTITY;

  SUBTYPE employee IS person
    ENTITY
      home_address : STRING (1 .. 50);
      office       : STRING (1 .. 8);
      phones       : SET OF STRING (1 .. 7);
      salary       : FLOAT;
      dependents   : INTEGER RANGE 0 .. 10;
    END ENTITY;

  SUBTYPE support_staff IS employee
    ENTITY
      supervisor : employee WITHNULL;
      full_time  : BOOLEAN;
    END ENTITY;

  SUBTYPE faculty IS employee
    ENTITY
      rank     : rank_name;
      teaching : SET OF course;
```

```
    tenure   : BOOLEAN := FALSE;
    dept     : department;
  END ENTITY;

SUBTYPE student IS person
  ENTITY
    advisor      : faculty WITHNULL;
    major        : department;
    enrollments  : SET OF enrollment;
  END ENTITY;

SUBTYPE graduate IS student
  ENTITY
    advisory_committee ; SET OF faculty;
  END ENTITY;

SUBTYPE undergraduate IS student
  ENTITY
    gpa  : grade_point := 0.0;
    year : INTEGER RANGE 1 .. 4 := 1;
  END ENTITY;

TYPE course IS
  ENTITY
    title    : STRING (1 .. 10);
    dept     : department;
    semester : semester_name;
    credits  : INTEGER;
  END ENTITY;

TYPE department IS
  ENTITY
    name  : STRING (1 .. 20);
    head  : faculty WITHNULL;
  END ENTITY;

TYPE enrollment IS
  ENTITY
    class  : course;
    grade  : grade_point;
  END ENTITY;

UNIQUE ssn WITHIN person;
UNIQUE name WITHIN department;
UNIQUE title, semester WITHIN course;
```

```
    OVERLAP graduate WITH faculty;

END university;
```

Figure A.1 University Database Schema

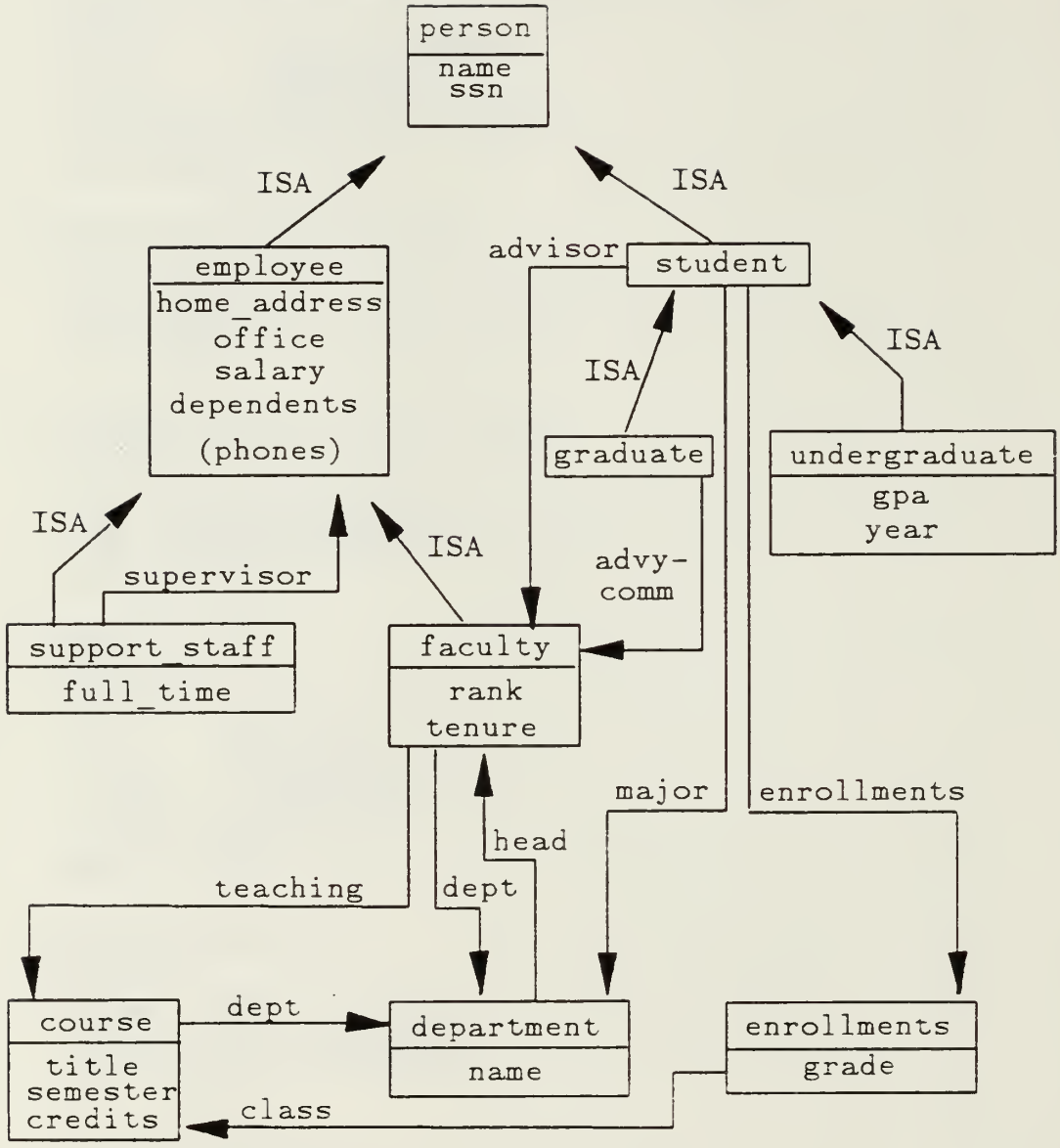


Figure A.2 Logical Graphical Representation of University Database Schema

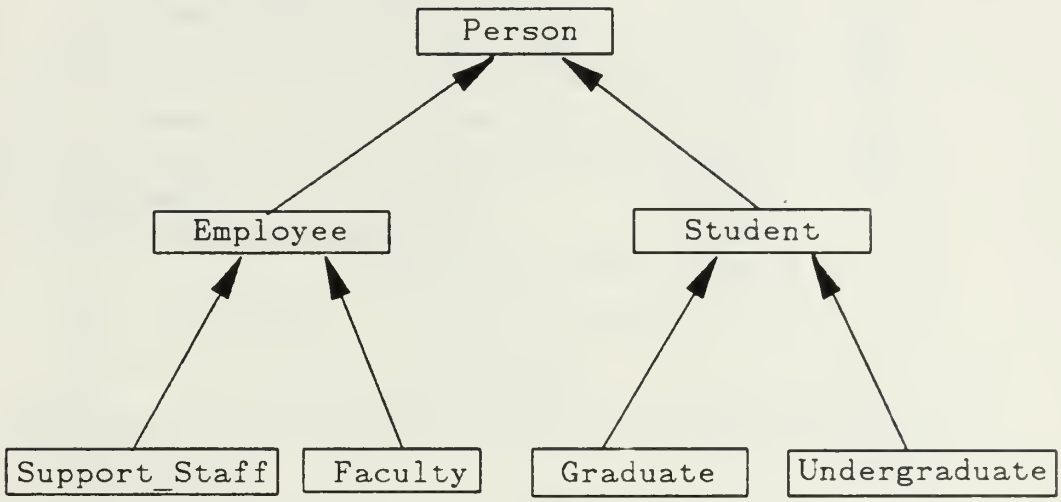


Figure A.3 Generalization Hierarchy of Person for the University Database Schema

```
struct function_node
  /* structure definition for each function type declaration */
      {
       char     fn_name[ENLength+1];
       char     fn_type;              /* either f(loat), i(nteger), s(tring),
                                      b(oolean), or e(numeration) */
       int      fn_range;             /* Boolean if range of values */
       int      fn_total_length;      /* max length */
       int      fn_num_value;         /* number of actual values */
       struct   ent_value    *fn_value;  /* actual value */
       struct   ent_node     *fn_entptr; /* ptr to entity type */
       struct   gen_sub_node *fn_subptr; /* ptr to entity subtype */
       struct   ent_non_node *fn_nonentptr; /* ptr to nonentity type */
       struct   sub_non_node *fn_nonsubptr; /* ptr to nonentity subtype */
       struct   der_non_node *fn_nonderptr; /* ptr to nonentity dertype */
       int      fn_entnull;          /* initialized false set true for no value */
       int      fn_unique;           /* init false - unique if true */
       struct   function_node *fn_next_fntptr;
      };

struct overlap_sub_node      /* list of ponters */
  /* structure definition for terminal subtypes that define one or more
     subtypes */
      {
       struct   gen_sub_node      *osn_name:  /* only terminal subtypes */
       struct   overlap_sub_node  *osn_next_name;
      };

struct overlap_ent_node     /* list of pointers */
   /* structure definition for subtypes with one or more entity supertypes */
      {
       struct   ent_node          *oen_name;
       struct   overlap_ent_node  *oen_next_name;
      };

struct gen_sub_node
  /* structure def for each generalization (supertype/subtype) node */
      {
       char     gsn_name[ENLength + 1];
       int      gsn_num_funct:    /* number of assoc. functions*/
       int      gsn_terminal;     /* if true (=1) it is terminal type */
       struct   overlap_ent_node  *gsn_entptr; /* ptr to entity supertype */
       int      gsn_num_ent;      /* number of entity supertypes */
       struct   function_node     *gsn_ftnptr;
       struct   overlap_sub_node  *gsn_subptr; /* ptr to subtype supertype */
```

```
        int     gsn_num_sub;    /* number of subtype supertypes */
        struct  gen_sub_node    *gsn_next_genptr;
       };

struct ent_node
  /* structure definition for each entity node */
       {
        char    en_name[ENLength + 1];
        int     en_num_funct;           /* number of assoc. functions */
        int     en_terminal;        /* if true (=1) it is terminal type */
        struct  function_node  *en_ftnptr;
        struct  ent_node       *en_next_ent;
       };



struct ent_value
  /* struct def for value of 'i','s','f','e', or 'b' */
       {
        char    *ev_value;      /* pointer to character string only */
        struct  ent_value       *ev_next_value;
       };

struct ent_non_node
  /* structure def for each base-type nonentity node */
      {
       char     enn_name[ENLength + 1];
       char     enn_type;                  /* either i(nteger), s(tring),
                                  f(loat), e(numeration), or b(oolean) */
       int      enn_total_length;          /* max length base-type value */
                                   /* of one string in ent_value */
       int      enn_range;          /* true or false depending on
                                   whether there is a range. If a
                                   range exists, there must be two
                                   entries into ent_value */
       int      enn_num_values;              /* number of actual values */
       struct   ent_value      *enn_value;  /* actual value of base-type */
       int      enn_constant;        /* boolean to refelect constant value */
       struct   ent_non_node    *enn_next_node;
      };

struct sub_non_node
  /* structure def for each subtype nonentity node */
      {
       char     snn_name[ENLength + 1];
       char     snn_type;                  /* either i(nteger), s(tring),
```

```
                                f(loat), e(numeration), or b(oolean) */
    int     snn_total_length;          /* max length of subtype value */
    int     snn_range;           /* true or false depending on
                                 whether there is a range. If a
                                 range exists, there must be two
                                 entries into ent_value */
    int     snn_num_values;              /* number of actual values */
    struct  ent_value     *snn_value;  /* actual value of subtype */
    struct  sub_non_node   *snn_next_node;
    };

struct der_non_node
  /* structure def for each derived type nonentity node */
    {
    char    dnn_name[ENLength + 1];
    char    dnn_type;                  /* either i(nteger), s(tring),
                               f(loat), e(numeration), or b(oolean) */
    int     dnn_total_length;      /* max length of derived type value */
    int     dnn_range;           /* true or false depending on
                                 whether there is a range. If a
                                 range exists, there must be two
                                 entries into ent_value */
    int     dnn_num_values;              /* number of actual values */
    struct  ent_value     *dnn_value;  /* actual value of derived type */
    struct  der_non_node   *dnn_next_node;
    };

struct ent_dbid_node
  /* structure def for each entity-relationship dbid node */
    {
    char    edn_name[DBNLength + 1];
    struct  ent_non_node   *edn_nonentity;
    int     edn_num_nonent;             /* number of nonentity types */
    struct  ent_node      *edn_entity;
    int     edn_num_ent;               /* number of entity types */
    struct  gen_sub_node   *edn_subptr;
    int     edn_num_gen;               /* number of gen_subtypes */
    struct  sub_non_node   *edn_nonsubptr;
    int     edn_num_nonsub;            /* number of nonentity subtypes */
    struct  der_non_node   *edn_nonderptr;
    int     edn_num_der;               /* number of nonentity derived types */
    struct  ent_dbid_node  *edn_next_db;
    };
```

```
If specified file not terminal entity
  abort CREATE routine.

While file not equal to specified CREATE file
Get next supertype | terminal file

  If expression = function_lookup IN
                supertype | terminal_type
    RETRIEVE ((File = supertype|terminal_type) and
            (function = lookup_value)) (file.key)
        | ((File = supertype|terminal_type) and
           (function = lookup_value)) or
     If entity_type
        INSERT(<File, new_entity_type>,
              <new_entity_type.key,**>,
              <db_function_name,lookup_value>,
              <database_function_names,
               values|pointers_to_values>)
     end_if
  end_if
  INSERT(<File,supertype|terminal_type>,
        <supertype.key|terminal_type.key, **),
        <database_function_names,
         values|pointers_to_values>)
END While
```

Figure C.1 Generalized Mapping Algorithm For CREATE

```
Take given filename and search entity types and
subtypes in entire database
  for function_expressions for filename.key
RETRIEVE follows from 1st 21 lines from assignment
if not one of entity types or subtypes
then
DELETE actual values or pointer values
```

Figure C.2 Generalized mapping algorithm For DESTROY

```
/* Generalized format of for_each_loop */

[RETRIEVE (File = Given_filename)(function.key)
 |    RETRIEVE (File = Given_filename)
                  (requested_function.key)]
[for each (function.key) V (requested_function.key)
  {RETRIEVE((File = function) and
       ((function.key|requested_function.key)=**)
   | and ((function op 'value') | (function op 'value')
                    and)) (desired_function)  END
 | RETRIEVE((File = function) and
          ((function.key|requested_function.key)=**)
   | and ((function op 'value') | (function op 'value')
                and)) (desired_function) BY
                                (function) END}
|  {RETRIEVE((File = function) and
          ((function.key|requested_function.key)=**)
   | and ((function op 'value') | (function op 'value')
                  and)) (desired_function)
  for each (function.key) V (requested_function.key)
  RETRIEVE((File = function) and
       ((function.key|requested_function.key)=**)
   | and ((function op 'value') | (function op 'value')
                and)) (desired_function)
                   BY (function)
    for each (function.key) V (requested_function.key)]
```

Figure C.3 Generalized Mapping Algorithm For FOR EACH Loop

```
 /* Generalized format of assignment_statement */

if (lookup_function IN given_filename)
then begin
  RETRIEVE ((File = given_filename) and
          (lookup_function = value))
          (given_filename_function.key)
end_begin
else begin
while (given_filename != terminal_subtype) and
     (given_function != function_name |
      descendant_function_name)
do
   RETRIEVE (File = descendant_file)
           (descendant_function_name(s))
end_while
while ((given_filename = terminal_subtype) and
     (lookup_function != function_name |
      ancestor_function_name))
     or ((descendant_file = terminal_subtype) and
     (lookup_function != function_name |
      ancestor_function_name))
do
   RETRIEVE (File = ancestor_file)
           (ancestor_function_name(s))
end_while
   RETRIEVE ((File = descendant_file | ancestor_file)
        and (lookup_function = value))
           (root_function.key)
end else_begin
if loop_function in ancestor_file to given_file
for each root_function.key | given_filename_function.key
[RETRIEVE ((File = Given_filename) and
  (given_filename_function.key = root_function.key |
                      given_filename_function.key))
         (given_filename_function.key)
end_if
for each (root_function.key |
         given_filename_function.key)
repeat
    /* find function */
{    for each indirect function expression
   if ((expression_filename) and
      (expression_function =
       expression_filename_function_name))
   then begin
```

```
    RETRIEVE ((File = expression_filename) and
            (expression_function = value))
            (expression_filename_function.key)
  end_begin
  else begin
  while (expression_filename != terminal_subtype) and
       (expression_function != function_name |
        descendant_function_name)
  do
    RETRIEVE (File = descendant_file)
            (descendant_function_name(s))
  end_while
  while ((expression_filename = terminal_subtype) and
       (expression_function != function_name |
        ancestor_function_name))
       or ((descendant_file = terminal_subtype) and
       (expression_function != function_name |
        ancestor_function_name))
  do
    RETRIEVE (File = ancestor_file)
            (ancestor_function_name(s))
  end_while
    RETRIEVE ((File = descendant_file |
                  ancestor_file) and
            (expression_function = value))
            (expression_filename_function.key)
  end else_begin
  if expression_function in ancestor_file to
    expression_file
  for each expression_filename_function.key
  [RETRIEVE ((File = expression_filename) and
          (expression_filename_function.key = *))
          (expression_filename_function.key |
           [aggregate_oper](actual_value))
  end_if
}
  if ((given_filename) and
    (given_loop_function =
     given_filename_function_name))
  then
    loop_function_file = given_filename
  end_if
  while (given_filename != terminal_subtype) and
       (given_loop_function != function_name |
                  descendant_function_name)
  do
```

```
    RETRIEVE (File = descendant_file)
            (descendant_function_name(s))
    loop_function_file = descendant_filename
  end_while
  while ((given_filename = terminal_subtype) and
       (given_loop_function != function_name |
        ancestor_function_name))
       or
       ((descendant_file = terminal_subtype) and
       (given_loop_function != function_name |
        ancestor_function_name))
  do
    RETRIEVE (File = ancestor_file)
            (ancestor_function_name(s))
    loop_function_file = ancestor_filename
  end_while
UPDATE ((File = loop_function_file) and
      (loop_file.key = root_function.key |
       given_filename_function.key) and
       (given_loop_function = actual_value |
        indirect_value))
   /* where indirect_value =
           expression_filename_function.key
    | [aggregate_operation](actual_value)  */
 /*and loop_function_file = ancestor_file |
          descendant_file | given_filename*/
if (given_loop_function != filename_function_name)
  then error;
if (given_loop_function and value are disjoint)
  then error;
until no_more_loop_functions
```

Figure C.4 Generalized Mapping Algorithm For ASSIGNMENT

```
    /* null value into existing set = error */
    /* duplicates just don't get added */

Copy 1st 21 lines from assign_statement
  if function_expression != actual value
    then do lookup of function expression value
  lookup:
       from INCLUDE
         RETRIEVE files on specified functions (may be
             more than one file to retrieve depending
             upon number and type of function values
             to key on from WHERE)
       from INTO
         if target function does not receive an actual
         value or target function does not receive its
         pointer value directly from a file in the
         INCLUDE, then before the pointer value can be
         INSERT into the target function, an INSERT
         will be required in the file pointed to by the
         target function to provide the required pointer
         value indirectly (see Example 1).
/* for check and unchanged values */
RETRIEVE given_filename
/* new variable names in code */
INSERT actual values or pointer values
```

Figure C.5 Generalized Mapping Algorithm For INCLUDE

```
    RETRIEVE (File = descendant_file)
           (descendant_function_name(s))
    loop_function_file = descendant_filename
  end_while
  while ((given_filename = terminal_subtype) and
       (given_loop_function != function_name |
        ancestor_function_name))
       or
       ((descendant_file = terminal_subtype) and
       (given_loop_function != function_name |
        ancestor_function_name))
  do
    RETRIEVE (File = ancestor_file)
           (ancestor_function_name(s))
    loop_function_file = ancestor_filename
  end_while
UPDATE ((File = loop_function_file) and
      (loop_file.key = root_function.key |
       given_filename_function.key) and
       (given_loop_function = actual_value |
        indirect_value))
   /* where indirect_value =
          expression_filename_function.key
    | [aggregate_operation](actual_value)  */
 /*and loop_function_file = ancestor_file |
         descendant_file | given_filename*/
if (given_loop_function != filename_function_name)
  then error;
if (given_loop_function and value are disjoint)
  then error;
until no_more_loop_functions
```

Figure C.4 Generalized Mapping Algorithm For ASSIGNMENT

```
    /* null value into existing set = error */
    /* duplicates just don't get added */

Copy 1st 21 lines from assign_statement
  if function_expression != actual value
    then do lookup of function expression value
  lookup:
       from INCLUDE
         RETRIEVE files on specified functions (may be
              more than one file to retrieve depending
              upon number and type of function values
              to key on from WHERE)
       from INTO
         if target function does not receive an actual
         value or target function does not receive its
         pointer value directly from a file in the
         INCLUDE, then before the pointer value can be
         INSERT into the target function, an INSERT
         will be required in the file pointed to by the
         target function to provide the required pointer
         value indirectly (see Example 1).
/* for check and unchanged values */
RETRIEVE given_filename
/* new variable names in code */
INSERT actual values or pointer values
```

Figure C.5 Generalized Mapping Algorithm For INCLUDE

```
if function_expression != actual value
  then do lookup of function expression value
lookup:
    from EXCLUDE
      RETRIEVE files on specified functions (may be
           more than one file to retrieve depending
           upon number and type of function values
           to key on from WHERE)
    from FROM
      if target function does not receive an actual
      value or target function does not receive its
      pointer value directly from a file in the
      EXCLUDE, then before the pointer value can be
      DELETE into the target function, a DELETE will
      be required in the file pointed to by the
      target function to eliminate the indirectly
      referenced pointer value (see Example 1).
        DELETE actual values or pointer values
          /* new variable names in code */
```

Figure C.6 Generalized Mapping Algorithm For EXCLUDE

```
If ((entity_valued_expression = entity_base_type)
 or (FROM entity_type_name = entity_base_type) or
    (INTO entity_type_name = entity_base_type))
            then abort MOVE
If (function_name =
        function_name_of_FROM_entity_type_name)
            then abort MOVE
If (entity_valued_expression would invalidate
     constraint in INTO entity_type_name) and
   (FROM entity_type_name !=
   entity_valued_expression)
            then abort MOVE
If ((entity_valued_expression =
     FROM entity_type_name)
    and (supertype_of_entity_valued_expression
       does not point to valid terminal subtype))
            then abort MOVE

If (File in entity_valued_expression) and
   (given search conditions are functions
    in entity_valued_expressions)
then
RETRIEVE ((File = file_in_entity_valued_expression)
         and
         (search_database_function_names =
          given values in entity_valued_expressions))
                  (file.key)
If (File in entity_valued_expression) and
   (given search conditions are not functions in
    entity_valued_expressions)
then
RETRIEVE (File = file_in_entity_valued_expression)
                              (file.key)
 until
RETRIEVE ((File=Corres_file_of_search_function_names)
         and (corres.key = file.key) and
         (search_database_function_names =
         given values in entity_valued_expressions))
                              (corres.key)
 /*              LOOP PARAMETER                */
If (entity_valued_expression =
   entity_valued_loop_parameter)
then
 for each (file.key | corres.key)
end_if
```

```
While (FROM entity_type_name(s)) and
        not terminal_subtype(s))
 do
   DELETE ((File = (entity_type_name(s) |
          terminal_subtype_name(s)) and
  ((entity_type_name | terminal_subtype).key =
                    file.key | corres.key))
end_while

While (INTO entity_type_name(s)) and
         not (terminal_subtype(s))
 do
 If (function_name_expression points to entity with
     pointer_value)
   RETRIEVE ((File = (entity_type(s) |
           terminal(s))_db_function_name) and
           (db_function_name =
            given value in db_function_expression))
     ((entity_type | terminal)_db_function_name.key)
 end_if
 INSERT (<File, (entity_type_name(s) |
              terminal_subtype_name(s)>,
        <(entity_type | terminal_type).key, **)>,
        <db_function_name,
         ((user | system)_default_value |
         (completely_specified user_value)
           | pointer to db_function_name(s)>
```

Figure C.7 Generalized Mapping Algorithm for MOVE

LIST OF REFERENCES

1. Fairley, R. E., *Software Engineering Concepts,* McGraw-Hill Book Company, 1985.
2. Demurjian, S. A., and Hsiao, D. K., "New Directions In Database-Systems Research And Development," Technical Report, NPS-52-85-001, Naval Postgraduate School, Monterey, California, February 1985.
3. Weishar, D. J., *The Design and Analysis of a Complete Relational Hierarchical Interface for a Multi-Backend Database System,* Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.
4. Macy, G., *Design and Analysis of an SQL Interface for a Multi-Backend Database System,* Master's Thesis, Naval Postgraduate School, Monterey, California, March 1984.
5. Wortherly, C. R., *Design and Analysis of a Complete Network Interface for a Multi-Backend Database System,* Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.
6. Rollins, R., *Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System,* Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.
7. Demurjian, S. A., et al., "Design Analysis And Performance Evaluation Methodologies For Database Computers," Technical Report, NPS-52-85-009, Naval Postgraduate School, Monterey, California, June 1985.
8. Benson, T. P., and Wentz, G. L., *The Design and Implementation of a Hierarchical Interface For The Multi-Lingual Database System,* Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.
9. Kloepping, G. R., and Mack, J. F., *The Design and Implementation of a Relational Interface For The Multi-Lingual Database System,* Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.
10. Emdi, B., *The Design and Implementation of a Network [Codasyl-DML] Interface For The Multi-Lingual Database System,* Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.
11. Anthony, J. A. and Billings, A. J., *The Implementation of an Entity-Relationship [Daplex] Interface For The Multi-Lingual Database System,* Master's Thesis, Naval Postgraduate School, Monterey, California, December 1985.
12. Demurjian, S. A., et al., "Performance Measurement Methodologies For Database Systems," Technical Report, NPS-52-84-023, Naval Postgraduate School, Monterey, California, December 1984.
13. Demurjian, S. A., et al., "The Implementation of a Multi-Backend Database System (MDBS): Part IV - The Revised Concurrency Control and Directory Management Processes and the Revised Definitions of Inter-Process and Inter-Computer Messages," Technical Report, NPS-52-84-005, Naval

Postgraduate School, Monterey, California, March 1984.

14. Kerr, D. S., et al., "The Implementation of a Multi-Backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS," Technical Report, NPS-52-83-008, Naval Postgraduate School, Monterey, California, June 1983.
15. Shipman, D. W., "The Functional Data Model And The Data Language Daplex," *ACM Transaction on Database Systems,* Vol. 6, No. 1, March 1981.
16. Gray, P. M. D., *Logic, Algebra And Databases,* Halsted Press, 1984.
17. Adaplex: Rationale and Reference Manual, CCA-83-08, Computer Corporation of America, Four Cambridge Center, Cambridge, Massachusetts, May 1983.
18. Fox, S., et al., "Daplex User's Manual, CCA-84-01," Computer Corporation of America, Four Cambridge Center, Cambridge, Massachusetts. June 1984.
19. Chen, P.S., "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transaction on Database Systems,* Vol. 1, No. 1, March 1976.
20. Ullman, J. D., *Principles Of Database Systems, Second Edition,* Computer Science Press, 1982.
21. MacLennan, B. J., *Functional Programming Methodology* Naval Postgraduate School, 1985.
22. Adaplex BNF, Computer Corporation of America, Four Cambridge Center, Cambridge, Massachusetts, May 1985.
23. Hsiao, D. K., et al., "The Use Of A Database Machine For Supporting Relational Databases," *Proceedings Of The 5th Annual Workshop On Computer Architecture For Non-numeric Processing,* 1978.
24. Hsiao, D. K., et al., "Performance Study Of A Database Machine In Supporting Relational Databases," *Proceedings Of The 4th International Conference On Very Large Databases,* 1978.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	2
4.	Curricular Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5100	1
5.	Professor David K. Hsiao, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
6.	Steven A. Demurjian, Code 52 Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	2
7.	Philip L. Goisman 3441 E. Edgemont St. Tucson, Az. 85716	3

Thesis
G529
c.1

Goisman
The design and analysis of a complete entity-relationship interface for the Multi-Backend Database System.